Introduction to Android

Mariusz Nowostawski
Department of Information Science
University of Otago, Dunedin, New Zealand

http://www.business.otago.ac.nz/infosci/

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

What is Android?

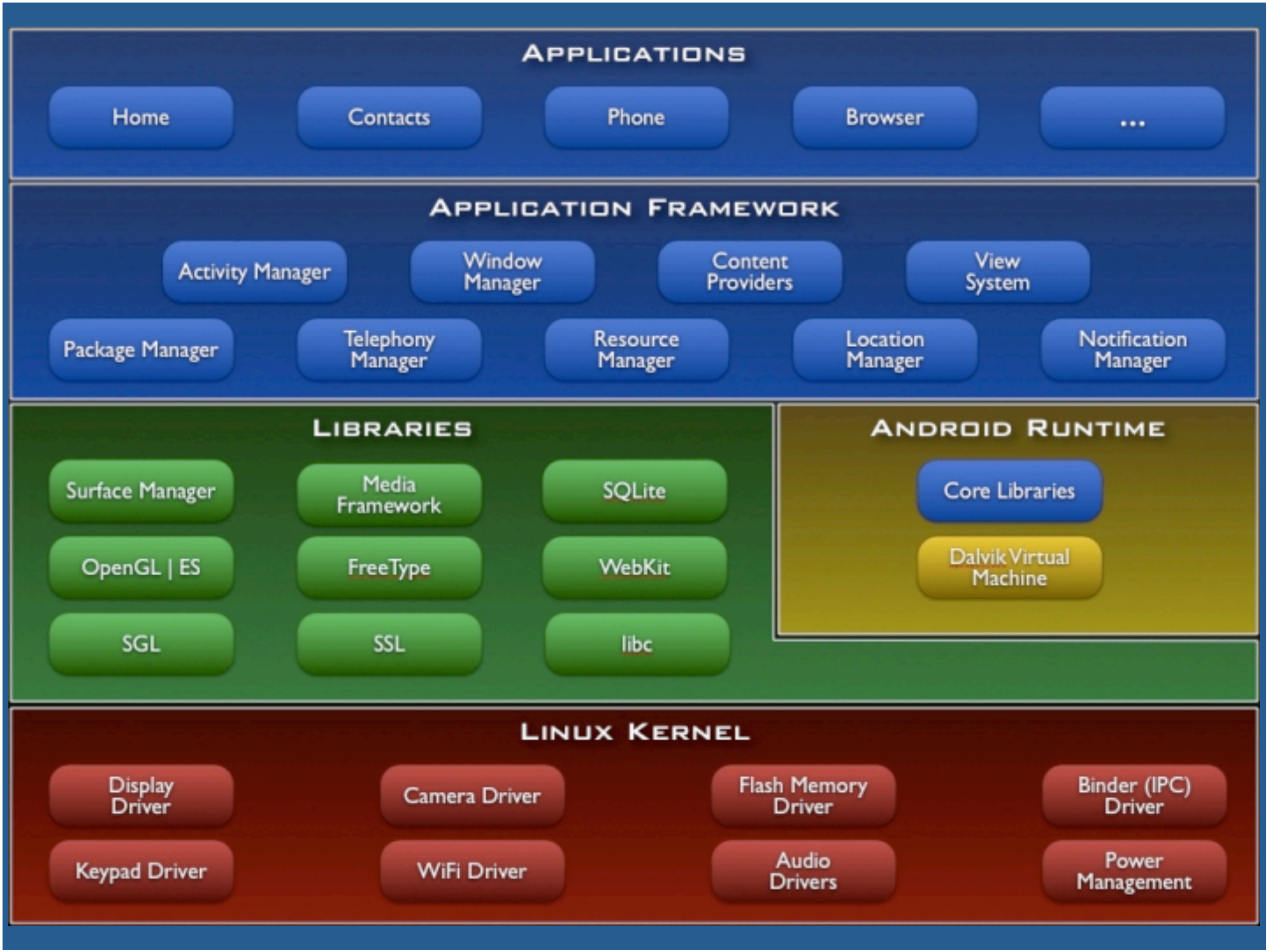
- A full software stack
  - Operating system
  - Middleware
  - Application framework
  - Key applications and services

Android: features

- Rich development environment: device emulator, tools for debugging, memory and performance profiling, plugin for the Eclipse IDE
- Application framework: enabling reuse and replacement of components
- Dalvik Virtual Machine
- Browser, optimized graphics: custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0 specification, SQLite for structured data storage, media support

Features

- Rich development environment: device emulator, tools for debugging, memory and performance profiling, plugin for the Eclipse IDE
- Application framework: enabling reuse and replacement of components
- Dalvik Virtual Machine
- Browser, optimized graphics: custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0 specification (hardware acceleration optional), SQLite for structured data storage, media support


Applications
Home
Contacts
Phone
Browser
...
Application Framework
Activity Manager
Window Manager
Content Providers
View System
Package Manager
Telephony Manager
Resource Manager
Location Manager
Notification Manager
Libraries
Surface Manager
Media Framework
SQLite
OpenGL | ES
FreeType
WebKit
SGL
SSL
libc
Android Runtime
Core Libraries
Dalvik Virtual Machine
Linux Kernel
Display Driver
Camera Driver
Flash Memory Driver
Binder (IPC) Driver
Keypad Driver
WiFi Driver
Audio Drivers
Power Management

The bottom layer

- Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model.
- The kernel acts as an abstraction layer between the hardware and the rest of the software stack.
- New open LIBC core

Application framework

- Full access to the same framework APIs used by the core applications.
- The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities.
- This mechanism allows components to be replaced by the developer/user.

Framework: core components

- no main() function
- stress on component reuse

- Activities
- Services
- Broadcast receivers
- Content providers

Activities

- Visual representations for user interactions
- Consists of a single or multiple Views
- A single application can have single or multiple activities
- Activities are typically independent

Services

- Represent tasks run on behalf of a user that do not have visual interface
- E.g. media player
- It is possible to connect to ongoing service
- Or start service when needed

Broadcast receivers

- Consume and react to system and application-level broadcasts
- E.g. battery low indication
- They do not have visual representation, but can start Activity in response to the broadcast
- Notification manager is a system-level handler for broadcasts

Content providers

- Make a specific set of the application's data available to other applications
- SQLite or filesystem can be used to stare persistent data
- Triggered through ContentResolver

Intents

- Activities, Services and Broadcast receivers are triggered through asynchronous messages: *INTENTS*
- For activities and services intents contain action name being requested and specifies the URI of the data to act on.
- For broadcast receivers, the Intent object names the action being announced. For example, it might announce to interested parties that the camera button has been pressed.

Activities and Tasks

- Task is a stack of activities. A sequence of consecutive activities run by the user.
- The activities may belong to the same or separate applications – for the user it will always appear as if the task forms a uniform application.
- E.g. navigation application.

Framework services

- Content Providers – enable applications to access data from other applications (such as Contacts), or to share their own data
- Resource Manager – provides access to non-code resources such as localized strings, graphics, and layout files
- Notification Manager – enables all applications to display custom alerts in the status bar
- Activity Manager – manages the lifecycle of applications and provides a common navigation backstack

Library support

- Libc subset, BSD-based, tuned for embedded use
- No Swing, no AWT, no standard Java packages
- Android-specific equivalents for everything
- Media Libraries, many media codecs
- Surface Manager - manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications
- 3D libraries - an implementation based on OpenGL ES 1.0 APIs; FreeType - bitmap and vector font rendering
- SQLite

Programming mobile devices

- Priority: battery life.
- Remember: anytime instruction is executed, it consumes power.

- Processing time scales
  - Human interaction scale: 10-30 interactions/sec
  - Human perception scale: 25-30 frames/sec
  - Computer scale: as much and as fast as possible

Rules of thumb

- Avoid computer scale computations (!)
- Spend most of its time sleeping
- React quickly and decisively to user and network input
- Avoid object allocations
- Short lived objects should be avoided because of GC costs. However - long lived objects take up memory. Trade-offs.

Android – Dalvik VM

- Work on Android has started in 2005
- Dan Bornstein – lead developer
  - experienced VM architect http://www.milk.com/home/danfuzz/
- 6 core developers plus contributors (libs)
- Dan developed the core of Dalvik VM
- Apache Harmony (subset) used for core libs

Dalvik – the name – the village

- Iceland's north-most fishing village
- 2000 inhabitants
- close to Arctic Circle

Dalvik – the name

Design objectives

- Slow CPU (initial target ARM architecture)
- Limited RAM
- OS does not have any swap space
- Takes battery life into consideration

CPU considerations

- Speed: 250-500 MHz
  - e.g. Qualcomm MSM 7200A 528 MHz
- Bus speed: 100MHz
- Data cache: 16-32kB

Memory considerations

- Memory efficiency top priority
- Design
  - total system RAM – 64 MB
  - Available RAM after low-level startup: 40MB
  - Available RAM after high-level services started: 20MB
- Multiple independent processes, separate address spaces, separate memory
- Added difficulty: large system library (10MB)

Bytecode optimisations

- Objectives:
  - Single constant pool, minimal repetitions
  - Per-type pools (implicit typing)
  - Implicit labeling

dx tool:

Java bytecode (.class) → Dalvik Executable (.dex)

Bytecode optimisations

- Results
  - on average DEX files are 50% smaller than uncompressed JAR files
  - on average DEX files are just slightly smaller than compressed JARs
- Why all the fuss then?

Other optimisations

- Byte swapping, padding (unnecessary on ARM)
- Static linking
- Inlining special native methods
- Pruning empty methods, e.g. constructor Object()
- Adding auxiliary data

Bytecode verification

- DEX structures cannot lie, and
- Code can't misbehave
- DEX structures: valid indices, valid offsets

- The verification is done mostly for the sake of bugs prevention (icing on the cake)
- Security is handled by … Linux.

No Just-in-Time (JIT) compiler

- Java and Dalvik designed mostly for user apps
- JIT wouldn't work well (warm-up time)
- JIT would put extra pressure on CPU/Mem
- Native code for libraries, graphics, media
- Hardware support: graphics, audio, etc
- For performance conscious there is JNI support

Dalvik memory model

- Clean memory (read only)
  - common DEX files (libraries and sys. libraries)
  - application-specific DEX files
- Shared dirty (mostly not written)
  - library 'live' DEX structures
  - shared copy-on-write heap
- Private dirty (read-write)
  - application heap, app. live DEX structures

GC design

- Considerations
  - Separate processes, heaps, and separate GCs
  - GCs must be independent
  - GC should respect the sharing
- Two generic strategies
  - GC data is stored together with the objects
  - GC data is stored separate from the objects

GC design

- Keeping GC data separate
  - Avoids un-sharing pages
  - Exhibits better small cache behaviour
  - Does not waste memory

Instruction set

- Java? Not quite. Not at all!
- Completely new instruction set
- Register machine (infinite register machine)
  - Avoids unnecessary memory accesses (tell how?)
  - Consumes instruction stream more efficiently
  - Higher semantic density per instruction (!)
- Result: 30% fewer instructions, 35% fewer code units, 35% more bytes in the instruction stream

Example

```
public static long sumArray(int[] arr) {
  long sum = 0;
  for (int i : arr) { sum += i; }
  return sum;
}
```

	bytes	dispatches	reads	writes
.class	25	14	45	16
.dex	18	6	19	6

Instruction dispatch

```
static void interp(const char* s) {
    for (;;) {
        switch (*(s++)) {
            case 'a': printf (“Hello"); break;
            case 'b': printf (“ "); break;
            case 'c": printf (“world!"); break;
            case 'd': prinf (”\n"); break;
            case 'e': return;
}}}
int main (int argc, char** argv) {  interp("abcbde"):  }
```

Computed GOTO (GCC way)

```
#define DISPATCH()  { goto *op_table[*((s)++) - 'a']; }

static void interp(const char* s) {
    static void* op_table[] =
        { &&op_a, &&op_b, &&op_c, &&op_d, &&op_e };
    DISPATCH();
    op_a: printf((”Hello"); DISPATCH();
    op_b: printf (“ "); DISPATCH();
    op_c: printf (”world!"); DISPATCH();
    op_d: prinf (”\n"); DISPATCH();
    op_e: return; }
```

Best dispatch implementation

- The computed GOTO can be further optimised if we re-write it in assembly.
- The code above uses typically two memory reads. We can lay out all our bytecodes in memory in such a way that each bytecode takes exactly the same amount of memory - this way we can calculate the address directly from the index.
- Added benefit is the cacheline warm-up for frequently used bytecodes.

References

- Dalvik on Wikipedia

  http://en.wikipedia.org/wiki/Dalvik_virtual_machine
- Android resources

  http://nzdis.otago.ac.nz/projects/wiki/android
- Google I/O 2008 Talk (Dan Bornstein)

  http://www.youtube.com/watch?v=ptjedOZEXPM

Thank you

mariusz@nowostawski.org
twitter: praeteritio
skype: nowostawski

http://www.otago.ac.nz/

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND