Cloud Computing: Theory and Practice

Mark Grechanik, Ph.D.

August 2, 2020

University of Illinois at Chicago
Department of Computer Science
851 S. Morgan Street
Chicago, IL 60607-7053
USA
drmark@uic.edu
https://www.cs.uic.edu/~drmark

Ordering Information: Quantity sales. Special discounts are available on quantity purchases by corporations, associations, and others. For details, contact the author at the address above.

Printed in the United States of America

Contents

1 Overview **5**
- 1.1 New Computing Metaphor . . . 5
- 1.2 The State of the Art and Practice . . . 7
- 1.3 Obstacles For Cloud Deployment . . . 9
- 1.4 What This Book Is About . . . 11

2 Duffusing 2 Cloud **12**
- 2.1 Diffusing Computation . . . 13
- 2.2 Peer-To-Peer (P2P) Computing . . . 15
  - 2.2.1 Backing up Files With Pastiche . . . 17
  - 2.2.2 <Project Name Here>@Home . . . 20
- 2.3 Grid Computing . . . 22
- 2.4 All Roads Lead to Cloud Computing . . . 23
- 2.5 Summary . . . 25

3 Model and Operations **27**
- 3.1 A Model Of Cloud Computing . . . 27
- 3.2 Cost And Utilization in Cloud . . . 30
- 3.3 The State of the Art And Practice . . . 31
- 3.4 Measuring Cloud Elasticity . . . 32
- 3.5 Cloud Service and Deployment Models . . . 36
  - 3.5.1 Software Stacks . . . 36
  - 3.5.2 Cloud Deployment Models . . . 37
  - 3.5.3 Software as a Service (SaaS) . . . 38
  - 3.5.4 Platform as a Service (PaaS) . . . 41
  - 3.5.5 Infrastructure as a Service (IaaS) . . . 42
  - 3.5.6 Function as a Service (FaaS) . . . 44
- 3.6 Summary . . . 46

4 RPC **47**
- 4.1 Local Procedure Calls . . . 47
- 4.2 Calling Remote Procedures . . . 51
- 4.3 The RPC Process . . . 55
- 4.4 Interface Definition Language (IDL) . . . 56

4.5 Pervasiveness of RPC . . . 58
4.6 Summary . . . 60

5 Map/Reduce Model **61**
5.1 Computing Tasks For Big Data Problems . . . 62
5.2 Datacenters As Distributed Objects . . . 64
5.3 Map and Reduce Operation Primitives . . . 66
5.4 Map/Reduce Architecture and Process . . . 67
5.5 Failures and Recovery . . . 69
5.6 Google File System . . . 70
5.7 Apache Hadoop: A Case Study . . . 73
5.8 Summary . . . 77

6 RPC Galore **78**
6.1 Java Remote Method Invocation (RMI) . . . 78
6.1.1 The Java RMI Process . . . 80
6.1.2 Parameter Passing . . . 81
6.1.3 Lazy Activation of Remote Objects . . . 82
6.2 <your data serialization format here>-RPC . . . 83
6.2.1 XML-RPC . . . 84
6.2.2 JSON-RPC . . . 87
6.2.3 GWT-RPC . . . 88
6.3 Facebook/Apache Thrift . . . 89
6.4 Google RPC (gRPC) . . . 90
6.5 Twitter Finagle . . . 91
6.6 Facebook Wangle . . . 93
6.7 Summary . . . 93

7 Cloud Virtualization **95**
7.1 Abstracting Resources . . . 95
7.2 Resource Virtualization . . . 97
7.3 Virtual Machines . . . 98
7.4 Hypervisors . . . 101
7.5 Interceptors, Interrupts, Hypercalls, Hyper-V . . . 104
7.6 Lock Holder Preemption . . . 106
7.7 Virtual Networks . . . 107
7.8 Programming VMs as Distributed Objects . . . 110
7.9 Summary . . . 111

8 Appliances **112**
8.1 Unikernels and Just Enough Operating System . . . 113
8.2 OS^v . . . 115
8.3 Mirage OS . . . 117
8.4 Ubuntu JeOS . . . 118
8.5 Open Virtualization Format . . . 119
8.6 Summary . . . 120

9 Web Services **123**
9.1 Simple Object Access Protocol (SOAP) 124
9.2 Web Services Description Language (WSDL) 127
9.3 Universal Description, Discovery and Integration 129
9.4 Representational State Transfer (REST) 132
9.5 Integrated Applications as Web Service Workflows 134
9.6 Business Process Execution Language 136
9.7 Summary 138

10 Microservices and Containers **139**
10.1 Microservices 140
10.2 Web Server Containers 142
10.3 Containers From Scratch 144
10.4 Docker 149
10.5 Kubernetes 151
10.6 Application Container Specification and rkt 154
10.7 Summary 155

11 Infrastructure **156**
11.1 Key Characteristics of Distributed Applications 156
11.2 Latencies 159
11.3 Cloud Computing With Graphics Processing Units 163
11.4 RAID Architectures 165
11.5 MAID Architectures 167
11.6 Networking Servers in Datacenters 168
11.7 Summary 171

12 Load Balancing **173**
12.1 Types of Load Balancing 174
12.2 Selected Algorithms for Load Balancing 175
12.3 Load Balancing for Virtual Clusters 178
12.4 Google's Borg and Omega 180
12.5 VM Migration 182
12.6 Load Balancers Are Distributed Objects 183
12.7 Summary 185

13 Spark Data Processing **186**
13.1 Spark Abstractions 187
13.2 Data Stream Operations 189
13.3 Spark Implementation 191
13.4 The Architecture of Mesos 193
13.5 Delayed Scheduling 194
13.6 Spark SQL 196
13.7 Summary 198

14 The CAP Theorem **199**
14.1 Consistency, Availability and Partitioning 200
14.2 Transactions and Strict Consistency Models 201
14.3 Two-Phase Commit Protocol 203
14.4 The CAP Theorem 205
14.5 Practical Considerations for CAP 207
14.6 Summary 210

15 Soft State Replication **211**
15.1 Consistency Models 212
15.2 Data Object Replication Models 214
15.3 Software Design of Highly Available Applications 215
15.4 Eventual Consistency. BASE. 217
15.5 Message-Queuing Middleware 221
15.6 Protocols For Convergent States 222
15.7 Case Study: Astrolabe 224
15.8 Summary 225

16 Facebook Cloud **226**
16.1 Facebook Developer Infrastructure 227
16.2 Measuring Existential Consistency 229
16.3 Configuration Management 231
16.4 Managing Performance of Web Services 233
16.5 The HipHop VM 235
16.6 Summary 237

17 Conclusions **238**

List of Figures

1.1 A Java-like pseudocode example of airplane seat reservation. 8

2.1 A clique of P2P network. 19
2.2 A pseudocode for a virtual CPU (vCPU). 24

3.1 A model of the cloud. 28
3.2 An example of the cost-utilization curve, where actual application usage of resources is shown with the thin (red) line and the allocation of resources by the cloud is shown using the thick (blue) line. Modified from source: amazon.com/ec2. 30
3.3 An illustrative example of the CUVE for a cloud-based application. The timeline of the operations is shown with the horizontal block arrow in the middle. The process starts with the customer who defines elasticity rules on the left and the events are shown in the fishbone presentation sequence that lead to the CUVE on the right. (With main contributions from Mssrs.Abdullah Allourani and Md Abu Naser Bikas) 34
3.4 Java-like pseudocode for a client that uses Google SaaS spreadsheet service. 39
3.5 Example of HTTP POST request to create a VM at a cloud hosted by Oracle. 43
3.6 Example of a response to the HTTP POST request to create a VM that is shown in Figure 3.5. 43

4.1 An illustration of the local procedure call. 48
4.2 A model of the client/server interactions when calling remote procedures. 52
4.3 The RPC process. Circles with numbers in them designate the order of the steps of the process with arrows showing the direction of the requests. 55
4.4 IDL code example of the authentication interface and its remote procedures. 57
4.5 The RPC development process with IDL. Circles with numbers in them designate the order of the steps of the process with arrows showing the direction of the requests. Dashed arrows show the interactions between the client and the server. 59

5.1 An illustration of the map/reduce model. 67
5.2 The architecture and the workflow of the map/reduce model. 68
5.3 Example of the Java skeleton code for the map/reduce in Hadoop framework. 75

6.1 A fragment of HTML code describing a list of items on a computer. . 84
6.2 A fragment of XML code describing a list of items on a computer. . . 84
6.3 A fragment of XML code describing a list of items on a computer. . . 85
6.4 Java-like pseudocode example of the XML-RPC-based implementation of the calculator with the single method `add`. 86
6.5 Scala pseudocode example of using futures to send messages to Twitter followers. 92
6.6 Scala pseudocode example of a Finagle's remote service. 92

7.1 VMM map. 101
7.2 Virtual memory schematic organization. 103
7.3 Hyper-V architecture. 106
7.4 Java pseudocode for moving a hard disk from one VM to another VM in VirtualBox. 110

8.1 Example of the OVF description of some VAP. 121
8.2 Java pseudocode for using VmWare API calls to export OVF of some VM. 122

9.1 An illustrative example of a SOAP request to call the method Add of the class WSrv that takes values of two integer parameters x and y. . . 125
9.2 A simplified example of mapping between WSDL elements and the source code of the Java class web service. 128
9.3 The UDDI interaction model. 130
9.4 Relationship between WSDL and UDDI. 131
9.5 A BPEL program that defines a synchronous RPC between a client and a server. 137

10.1 Scala pseudocode example of a RESTful web service using Twitter Finch. 143
10.2 The skeleton C program of a container management system. 146
10.3 The dependencies of the program `/bin/sh` are obtained using the utility `ldd`. 147
10.4 An example of Dockerfile for creating a container with a Java program. 149
10.5 The skeleton C program of a container management system. 150

11.1 C program fragment for iterating through the values of the two-dimensional array. 161
11.2 A high-level view of the GPU architecture and interactions with the CPU. 164

12.1 A Java-like pseudocode for using a load balancer with Amazon Cloud API. 184

13.1 A Spark program for computing the average value for the stream of integers. 192
13.2 An SQL statement that retrieves information about employees. 197
13.3 A Spark SQL statement that retrieves information about employees. . 197

14.1 An illustration of a three-tier application. 200
14.2 The context of the CAP theorem. 206

15.1 A sales ACID transaction that updates data in three database tables. . 220
15.2 A transformed sales transaction that sends messages to update data. . 220
15.3 Pseudocode shows possible synchronization. 221
15.4 A transformed sales transaction that sends messages to update data. . 222

16.1 Loop formation. 230
16.2 HHVM tracelet that compares two input values. 236

Chapter 1

Overview of Key Drivers of Cloud Computing

This book is about designing, writing, testing, debugging/troubleshooting, and maintaining the code of computer programs whose components or objects communicate with each other and they are run in cloud computing environments where additional resources are allocated to these applications when the demand for their services increases and release resources when the demand decreases to ensure excellent quality of service and low cost of deployment. A fundamental difference between cloud computing and many other forms of computing (e.g., distributed, grid computing, or peer-to-peer (P2P) computing) is that resources such as CPUs and RAM can be added on demand to improve the performance of the application that is deployed in a cloud environment and the application owners pay for the usage of these resources. It is analogous to a user opening the cover of her smartphone or laptop, finding an available slot for a CPU and plugging in a new CPU or a memory card while continuing to run applications and paying only for the portion of time during which this CPU or the memory are used and then removing this CPU or the memory card when they are not needed any more. Of course, few users would manipulate their computing devices in this manner, however, since software applications are made available to millions of users on the first day of their deployment, it is more important than ever to guarantee that these applications provide services with excellent performance, among other things.

1.1 New Computing Metaphor

A modern fairy tale of large-scale software development features Captain Hook as a stakeholder of a software project who steals magical seven league boots from a fairy to make his applications leap great distances across their computational landscapes. Seven league boots is a metaphor for *cloud computing*, a model for enabling these applications to leap great distances magically, i.e., to access via networks a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) ubiquitously and on-demand, so that these resource can be rapidly provi-

sioned and released with minimal management effort [117]. Since performance and availability are very important non-functional characteristics of large-scale software applications, the promise of cloud computing is to enable stakeholders to economically achieve these characteristics via cloud elasticity and economy of scale. Essentially, a main benefit of deploying large-scale applications in the cloud is to significantly reduce costs for enterprises due to cheap hardware and software in the centralized cloud platforms. Thus, a main reason for enterprises to move their applications to the cloud is to reduce the increasing cost of their maintenance that involves paying for hardware and software platforms and technical specialists that maintain these platforms [13, 23, 24, 93, 98, 123].

Seven league boots is a metaphor for describing the operational efficiency of cloud allocating hardware and software services to deployed applications, which is in theory more optimal with increasing scale. This operational efficiency is often referred to as the *elasticity* to provision resources rapidly and automatically, i.e., to quickly *scale out*, and rapidly release these resources, i.e., to quickly *scale in*. Using our analogy, one can imagine a magical mini-robot who lives inside computer servers and who adds the CPUs and other resources when applications need them and removes these resources when they are no longer needed. This magical mini-robot is implemented in software that is a part of a cloud computing platform.

The difference between the words allocation and provisioning of resources is subtle but important. Allocating resources means reserving some quantity of these resources for the use of some client or a group of clients. For example, an organization can purchase 1,000 virtual machines (i.e., simulated computers) with some predefined characteristics from a cloud provider for 24x7 use for three years and the cloud provider will allocate these resources. The organization may have many users who will execute various applications in the cloud, and each of these applications will demand different resources. Provisioning resources to applications means activating the subset of the allocated allocated resources to meet the applications' demands. Ideally, stakeholders want to allocate exactly as many resources as will be provisioned to their applications, however, it is often difficult to achieve.

To stakeholders, the capabilities available for provisioning often appear to be unlimited, but unlike seven league boots, stakeholders purchase certain quantities of these capabilities [117]. A message from cloud providers (e.g., Google, Microsoft, Amazon EC2) to application owners is the following: develop and test your software application the same way you have been doing, and when you are ready, put on your application our magical seven league boots, i.e., deploy your application on our cloud platform, pay fees for the usage of our resources by your application, and we will take care of the performance and availability of your application. Despite this simple and attractive marketing message, companies and organizations are very slow to deploy their applications on the cloud; in fact, many companies that attempted to move their applications to cloud provider platforms moved back to internal deployment.

Not only is it widely documented that companies and organizations go slow with cloud computing, but they are also advised to hold back and thoroughly evaluate all pros and cons. A survey of 1,300 information technology professionals shows that it takes at least six months to deploy an application in the cloud and 60% of respondents said that cloud application performance is one of three top main reasons to move ap-

plications to the cloud, whereas 46% cited the cost of moving to the cloud as the major barrier [36]. Interestingly, almost two in five of respondents said they would rather get a root canal, dig a ditch, do their own taxes than address serious challenges associated with cloud deployments. Over 31% said they could train for a marathon or grow a mullet in a shorter period of time than it takes to migrate their applications to the cloud. There is a fundamental disconnect between software development process and deploying applications in the cloud, where properties of the cloud environment affect not only the performance of deployed applications, but also their functional correctness. In addition, more than one quarter of the respondents suggested that they had more knowledge on how to play Angry Birds than the knowing how to migrate their company's network and applications to the cloud.

1.2 The State of the Art and Practice

Some time ago, a large consulting company with over 250,000 employees worldwide organized a retreat for representatives from several dozens of Fortune 100 companies that included software managers from financial organizations, major insurances, and biomedical and pharmaceutic companies. Collectively, these managers had over 2,500 years of software development and maintenance experience. During the retreat, two hours were allocated for a topic on cloud deployment, but it finished in less than 20 minutes. The answer from most managers was that there was not much to discuss, that the idea of cloud computing was great, but they tried it, and it did not work for their software applications, so they went back to then proven in-house deployment. Private interviews revealed a pattern that emerged from these failed attempts.

Consider a flight reservation web-based application where a user chooses a flight and then reserves a seat on the plane as it is shown in Figure 1.1. The implementation is done using the class `ReservationSeatPlane` that is instantiated within the flight reservation application and it starts a thread that receives a request from users in the method `run` that calls the method `FindAndReserveSeat` that assigns a seat for a particular user. The Java keyword `synchronized` tells the runtime to use the internal locking mechanisms to synchronize accesses from multiple threads to objects that represent seats when executing the method `FindAndReserveSeat`, so that only one user can be assigned to a particular seat. The code was deployed as a part of a flight reservation application at a large corporation at a large server that handled multiple user requests. As the number of users increased, the corporation kept purchasing a bigger and more powerful server with more CPU cores and RAM to ensure that the response time of the application can be kept within the desired constraints.

At some point, a decision was made to stop purchasing computer hardware and invest into cloud deployment, since not only it would provide sufficient resources on demand to cope with user peak loads, but also it would allow the corporation to reduce the number of support personnel (i.e., system administrators and technicians) who troubleshoot and maintain the hardware. The binaries for the application were transferred to the cloud without any changes to the source code and deployed in a *virtual machine (VM)*, a simulated computing environment where some resources are emulated. We will explain the definition later in the book.

```
1 public class ReservationSeatPlane implements Runnable {
2 public synchronized void FindAndReserveSeat(User userID){
3         //finds a seat and assigns it to userID
4         }
5 public void run(){FindAndReserveSeat(user.getUserID());
6 }  }
```

Figure 1.1: A Java-like pseudocode example of airplane seat reservation.

Shortly after this deployment a disturbing pattern emerged that the customer support center of the corporation received an increasing number of calls from upset customers who were denied boarding to their flights, since some other customers were already in their seats. Even though denied boarding is a more humane solution than dragging passengers off their seats by security officers, passengers could not appreciate their luck and they complained loudly. As it turned out, the flight application issues multiple tickets for the same seats. Further investigation showed that the increased seat double-booking happened on flights that offered last-minute special deals. When the application was moved back to the corporation and deployed on a single powerful server, this pattern of double-booking disappeared. Within the corporation the consensus was formed that was succinctly expressed as "cloud computing is terrific but it is not for us yet."

A problem with this deployment was uncovered later during one of code reviews. The Java keyword `synchronized` guarantees that only one client can access the shared state as long as the threads run within a single process. However, once the load increases, the cloud scales out by adding more VMs that run threads in separate processes. Thus, two or more threads are prevented to reserve the same seat within the same process, however, the state of the application is not shared among two or more processes. Hence, double-booking occurred frequently when a special flight deals were announced leading to the peaks in user loads, which resulted in the cloud scale-out and creating new VMs. Preventing this situation would require changes to the source code, so that multiple VMs would synchronize their accesses to shared resources (i.e., seats), which is often an expensive and error-prone procedure.

An overarching concern was that most software applications have performance problems, i.e., situations when applications unexpectedly exhibit worsened characteristics for certain combinations of input values and configuration parameters. Performance problems impact scalability thus affecting a large number of customers who use a software application. To ensure good performance, the cloud is supposed to elastically allocate additional resources to the application to cope with the performance problems and the increased load. Predicting performance demands is a very difficult problem, and since applications are deployed as black-boxes, it is difficult for the cloud to allocate resources precisely when demand goes up and down. Unfortunately, the cloud has no visibility into the internals of the applications and their models, and resources are under- or over-provisioned resulting in reduced quality of services or significant cost of resources that are not used by these applications. Application owners ultimately foot the bill for these additional resources, which is considerable for large-scale applications with performance problems.

1.3 Obstacles For Cloud Deployment

There are two main reasons why provisioning cloud resources precisely to applications is a hard and open problem. First, cloud infrastructures are general tools that provide services to applications on demand, making it infeasible to predict all future needs for resources statically. Second, and more importantly, cloud infrastructures are designed to execute applications efficiently, and imposing runtime analysis for predicting future resource usage by these applications adds significant overhead. In short, cloud infrastructures cannot provision resources with a high degree of precision for the same reason that the schedulers of operating system kernels do not preempt processes in a way to optimize allocation of resources to processes precisely – it is not feasible to find the optimal context switching schedule quickly for multiple processes and the overhead of doing it is prohibitive. Consider a situation when a portion of the CPU time will be used to optimize the allocation of the CPU to different processes. In the worst case, doing so will consume the majority of the CPU with little capacity left to process applications.

An ideal solution is to supply a *behavioral model* of the application (i.e., a collection of constraints, performance counters, and various relations among components of the application [91, 127]) to the cloud infrastructure, so that it can provision resources to this application with a high degree of precision using this model. Behavioral models of applications make it easier for software engineers to find performance problems [19, 72, 118] by pinpointing some of these behaviors that involve intensive computations that are characteristic of performance problems. Essentially, a behavioral model of an application is a function that maps values from the domain of inputs and configuration parameters to the range of performance characteristics. For example, we obtained a behavioral model for a web store application as part of our preliminary work, and this model specifies that when 1,000 customers log into the application and approximately 60% of these customers proceed with the purchase while 40% of them browse items in some categories, the web application loses its scalability and its certain components are likely bottlenecks, which are phenomena where the performance of the application is limited by one or few components [4, 11]. Obtaining this model as part of performance testing leads to exploring how adding cloud resources can enable the application to scale under these conditions.

Unfortunately, behavioral models that summarize the runtime behavior of applications and their performance characteristics are difficult to obtain. Depending on input values, an application can exhibit different behaviors with respect to resource consumption. A big and important challenge is to obtain behavioral models for nontrivial applications that have a very large space of input parameter values. A precise model can be obtained if an application is run with all allowed combinations of values for its inputs. Unfortunately, this is often infeasible because of the enormous number of combinations; for example, 20 integer inputs whose values only range from zero to nine already leave us with 10^{20} combinations. Knowing what combinations of test input data to select for different input parameters to obtain acceptable behavioral models is very difficult. Thus, a fundamental problem of effective and efficient provisioning of resources in the cloud is how to obtain behavioral models of underconstrained applications with high precision as part of software development process. A related problem is how to use a behavioral model of an application to make the cloud infrastructure to

precisely provision resources to this application.

Consider a situation when a company develops a new software application and decides to test it in the cloud. Two main characteristics that the company measures as part of testing is the statement coverage of this application (i.e., how many statements or lines of code are executed as the percentage of the total number of statements as a result of running tests) and the number of reported failures. As in any software development activity, certain budget is allocated for software testing with constraints on resources and time. For example, it is expected that 70% to 80% statement coverage should be achieved in one month of testing. Of course, to achieve this coverage, test engineers must know how to generate test input data that will lead to executions of certain statements. For linear software applications, which do not contain any branch or loop statements, any input data would reach 100% statement coverage as long as no exceptions are thrown. However, with loops, branches, pointers, virtual dispatch, and other programming constructs, the task of generating test input data to increase test coverage becomes undecidable.

As a result, the company puts the application in the cloud and starts running multiple instances of this application in parallel in many VMs. Let us assume that the cost of the VM is one cent per hour and the average elapsed execution time of the application is one minute. Suppose that we run 1,000,000 VMs in parallel at the cost of $3.36Mil performing over 20Billion test executions of this application. Let us also assume that that the total input space is 10^{20} combinations. Yet, despite this cost, only a small fraction of the input space, 2×10^{-10} is explored. Of course, depending on the structure of the application's source code and its functionality, it may be enough, however, it is also highly probable that after spending two weeks of computing time and millions of dollars, the application's coverage may still be in single digits with few bugs reported. As a result, the testing effort in cloud would be declared as a failure, since the management can argue that by purchasing computers for the internal information technology (IT) division is more cost-effective that using cloud computing. And they may be right – these computers can be used for other applications, whereas the money paid for the cloud computing time used for the testing effort cannot be amortized for other applications.

This example illustrates complex issues that accompany cloud computing. Owning computing infrastructure is expensive, however, its cost is amortized over many applications, whereas the cost of poorly executed cloud deployment cannot be recouped. Moreover, depending on how confidential data is, i.e., public, partly or fully sensitive, it may not be possible to move it outside the organization that owns this data to the cloud infrastructure owned by a different company. Even though we do not discuss security and privacy of cloud deployment in this book, this is one of the biggest obstacles in cloud deployment. Thus, it is important to understand how to build an application as a set of distributed objects and deploy them in the cloud environment in a way that is economical, to guarantee desired performance and functionality.

1.4 What This Book Is About

This book fills a gap between the theory and practice of software engineering and deployment of applications in the cloud, where the cloud is viewed as an execution platform rather than a part of software development life cycle. Performance problems that go unnoticed during development and testing of software applications result in serious degradation of the quality of service that these applications should deliver to customers in the cloud. With global spending on public cloud services exceeding $110Bil [12, 58] and with an estimated $20Bil of the US federal government spending on cloud computing [84], the cost of engineering poor quality software for the cloud is measured in billions, making it a big and important problem of software engineering for the cloud [14].

This book describes abstractions, concepts, strategies, platforms, and techniques for engineering distributed objects for cloud computing. We do not use the word "object" in a pure object-oriented sense as a first-class dynamically dispatched behavior or an instance of a class, but rather as a service concept, where an object is a logical computing unit with defined physical boundaries that receives requests from its clients, performs computations, and responds when necessary. All communications between clients and objects are accomplished via sending sequences of bytes in predefined formats that we call messages using some protocols. A main perspective in this book is that everything is an object. A cloud is an object that its clients can view as a virtual computer. That is, the complexity of the cloud infrastructure is hidden from the user, who interacts with the cloud via the Internet as if this cloud is a standalone computer. The cloud runs VMs, which are objects themselves that contain many other objects that communicate with one another to implement some algorithms, workflows, or protocols. The user itself is an object that communicates with cloud and its objects by sending messages and receiving them from objects in the cloud.

Chapter 2

From Diffusing Computation To Cloud Computing

In this chapter, we describe the ideas of diffusing computation, peer-to-peer (P2P) computing, and grid computing as most notable precursors to cloud computing. All these types of computing are instances of distributed computing, which can be defined as software processes that are located at different memory address spaces and they communicate by passing messages, which are sequence of bytes with defined boundaries. These messages can be exchanged *synchronously*, when the message sending component blocks during the message exchange, or *asynchronously*, when the function call to send a message returns immediately after passing the parameter values to the underlying message-passing library and the component continues execution. A synchronous call may block during the entire call or a part of the call whereas an asynchronous call is always nonblocking. If the call returns before the result is computed, the caller must perform additional operations to obtain the result of the computation when it becomes available. Distributing computations is important for various reasons that include but not limited to better utilizing resources and improving response time of the application.

A stricter definition of synchronicity in distributed applications involves the notion of the global clock. In an idealized distributed application, all of its objects have access to this clock that keeps the time for these objects. All algorithm executions are bounded in time and speed and the global clock measures these time and speed. All message transmissions have bounded time delays (i.e., latencies). To understand these constraints, let us view the execution of some algorithm in phases – each phase groups one or more operations and there is a precisely defined phase transition that specifies when one phase finishes and the next one starts [132]. Suppose that we have a specification that defines at what phases to stop the executions of algorithms in some distributed objects and at what phases to allow the executions in some other distributed objects to proceed. Since partitioning executions in phases bounds their speeds and having the global clock bounds their times, it is possible to implement specifications that determine how a distributed application is executed (e.g., how resources are shared among its distributed objects). However, it is not easy to realize in the asynchronous

application when there is no global clock and no bounds exist on the execution time and speed of its constituent distributed objects.

Opposite to engineering distributed applications is building a monolithic application, whose entire functionality is implemented as instructions and data that are embedded in a single process address space. Since a single address space is often associated with a single computing unit, there is a limit on how many applications can be built as monolithic. Due to geographic distribution of users and data and government regulations, many applications can only be built as distributed. A key element in building distributed application is to organize objects that constitute these applications into some patterns, and then use these patterns to develop the underlying framework for certain types of distributed applications. In this chapter, we analyze several salient developments on the road from building single monolithic applications to creating and deploying applications in the cloud from distributed objects.

2.1 Diffusing Computation

One of key notions of organizing computing nodes into a graph structure was proposed by Dijkstra and Scholten [45]. Each node in the graph is a computing unit or an object, which can send and receive messages from other objects in this graph. The edges designate relations among the nodes; incoming edges to nodes designate messages that these nodes receive from other nodes, for which these edges are outgoing. In *diffusing computation*, some parent nodes send messages along the outgoing edges to their children nodes, who upon receipt of the messages send other messages to their children nodes respectively. Each node can receive messages not only from its parents, but also from its children; we say that children can signal their parents about completions of the computations. Thus, computation messages go from parent to children nodes and signals about computations go the other way around. A constraint is defined that each node can send a message only once. That is, the computation diffuses in the graph of the nodes, where each node performs a part of the computation and sends results to the other nodes that aggregate these results.

One can envision the realization of diffusing computation as a messaging service like Twitter where nodes are individual computing devices (i.e., objects) that receive a message and resend it to other device addresses, which are stored on each device.A question is if this diffusing computation terminates, since one can imagine that the nodes keep sending and receiving messages forever once this process is initiated.

A proof is defined by introducing a concept of deficit that is defined for each edge as the difference between the number of sent messages and received signals. Each node is also assigned the sum of deficits of its incoming edges and the value of the deficit for each node will always be greater or equal to zero. Initially, the edge and node deficits are zeros; once a node sends a message along the outgoing edges, the deficits of these edges are incremented, so sending messages never reduces deficits. Once a node received and processed a message, it sends a signal back to its parent, which decrements the deficit of the sender. At some point, all nodes will be reached including those that do not have any children and those that already sent a message. Once the computation terminates, messages and signals are not sent any more, and the sum of

the deficits of incoming edges will be zero. Since the sum of the deficits of all outgoing edges is equal to the sum of the deficits of the incoming edges, after a bounded number of steps, the computation will return to its neutral state and it is terminated.

An important contribution of the concept of diffusing computation is that it introduced an abstraction of cooperating distributed objects that receive service requests from other nodes. Diffusing computation is abstract, since no concrete details of the message formats, interfaces of computing units, or specific resources assigned to units are given. As such, diffusing computation offers a simple model that can be instantiated in a variety of concrete settings. For example, consider a simplified idea of a computation, which we later call *map/reduce*, where objects are assigned to different computing nodes and the dataset is distributed across a subset of these nodes. The objects that are run on these nodes will preprocess assigned datasets by mapping their data to smaller datasets, and send the smaller datasets to children nodes that will continue the computation until the data is reduced to some value, after which, these nodes will return these values to their parents as signals of completed computations. The parent nodes will aggregate these values and send them as signals to their parents until the computation terminates.

Question 1: Explain how to implement a diffusing computation model using a graph with cycles rather than trees.

A simple form of the diffusing computation is a graph of two nodes where one node called a *client* sends a message to the other node called a *server*. This communication may be synchronous, where the client will wait, i.e., will *block* until the server responds to the message or until some event is produced during the execution of the call by the server, or asynchronous, where the client will continue its computation while the server is processing the message. This model is called client/server and it is widely used to engineer distributed software applications. Technically speaking, this model is also valid for monolithic applications, where a client object invokes methods of some server objects. This invocation may be thought of the client object sending a message to the server object, even though in reality the invocation is accomplished by loading the instruction pointer with the address of the code for the invoked method, since the code is located in the same address space with the caller object. However, if the address spaces are disjoint, which is the case even when the client and the server are run in separate process spaces on the same computer, then the invocation is performed using some kind of inter-process communication mechanism (e.g., shared memory, pipes, sockets), which can be modeled using message passing.

Question 2: Can purchasing an item from a web store be explained in terms of diffusing computation?

Finally, using the model of diffusing computation, one can reason about splitting a monolithic computation into distributed one, so that it can meet various objectives. Computing nodes in the graph can be implemented as commodity server computers;

edges represent network connectivity among these computers; messages designate the input data that are split among computing nodes, and signals are output data that the servers compute. Adding constraints to the model of diffusing computation, we can create interesting designs, for example, for a web store, where purchasing an item triggers messages to various distributed objects, which are responsible for calculating sales tax, shipment and handling, retrieving coupons, among other things. Each of these objects may send messages to its children objects to trigger subsequent operations, for example, the shipment and handling component may send messages to components that compute an optimal shipping route and determine discounts for different shipping options. Once these computations are completed, children components send results of their computations with signal messages to their parents, who in turn do the same for their parent objects until the computation terminates. Thus, diffusing computation is an important model for understanding how objects interact in a distributed environment.

2.2 Peer-To-Peer (P2P) Computing

In 1999, company called Napster released a version of a file sharing system that was customized for audio file sharing over the Internet. Participants of Napster acted as autonomous entities who connected and left at will and whose goal was to exchange audio files with other participants. A generalization of this implementation is a graph where nodes represent participants or peers and edges describe the connections between peers. Whereas clients request services from servers in the client/server model, in peer-to-peer (P2P) model, each peer can request and receive services from any other peer. That is, each peer can act both as a client and a server. Sharing files is an instance of sharing resources, since files occupy space and thus the storage is shared. Thus, a question is whether a P2P network can be viewed as a computing cloud where resources are shared among participants who obtain necessary resources on demand and pay for them.

Since there is no central authority that dictates how resources are shared and peers join P2P networks anonymously, provisioning resources on demand is a very difficult problem. *P2P freeloading* is a term that designates peers who only receive services but never provide them. Since peers do not pay for services, they have incentives to capture as many resources as possible without sharing any of their own. In addition, once a peer starts providing services, there is no guarantee that the peer will continue these services or even stay connected to the network. As a result, services are not guaranteed in general and some P2P networks collapsed because of the freeloading behavior of peers.

Despite these serious drawbacks of P2P distributed computing, it poses various important questions that are instrumental in addressing various issues in building large-scale software applications from distributed objects. First, how to store and locate data efficiently in a large network of distributed objects with no central control? An insight into addressing this problem is that not all nodes should be connected to all other nodes (i.e., there is no need to form a clique), but rather than small subgroups of peers are organized into virtual nodes (i.e., these nodes are not physical entities, hence the name virtual). At a low-level network layer, peer objects communicate with one

another using a networking protocol like TCP/IP, whereas at the peer layer the objects communicate with one another using the high-level communication graph, which is called an *overlay network*.

Question 3: Discuss a model of P2P with a central computing unit that allows P2P nodes to organize into an overlay network by sharing information about their services with the central computing unit.

Second, the issue of security is important, since peer computers are exposed to different clients, some of which have malicious goals. A main idea behind many attacks is for one malicious users to mask as a peer to take over as many computers on the P2P network as possible for various purposes like stealing personal data, money, or to mount a bigger attack. Denial of service is one example of an attack when a malicious computer floods other computers with messages, so that they allocate an increasing chunk of their computational resources to processing these messages. Interestingly, the peer who initiates this process may not even be malicious, since the process can start with broadcasting a message across a P2P network to which the increasing number of peers send response messages. Thus, the diffusing computation is aggravated into a broadcast storm, where the network traffic is saturated with response messages and new messages are blocked. As a result, the entire P2P network becomes nonresponsive, a situation that is referred to as network meltdown. Detecting and preventing security threats is a big problem with P2P computing.

The third category of important questions that P2P computing raises and that is highly relevant to cloud computing is how to guarantee certain properties of computations in presence of failures. Unlike client/server networks where the failure of the server stops computations, there is no single point of failure in P2P networks, since handling peer failures should be engineered by design. In the worst case when only two peers are present, P2P degenerates into the client/server computing, however, as a large number of peers join the network and do not leave suddenly at the same time, single peer failures will not jeopardize the entire network.

Question 4: Suppose that a music player has a security vulnerability that can be exploited by sending a music file with carefully crafted sequence of bytes. Explain how you would design a P2P music file distribution system to mitigate the effect of such security vulnerability.

P2P applications are diverse and music sharing is what made P2P a part of the widespread cultural significance. We consider two case studies of applications of P2P to rather somewhat less general and more concrete problems of backing up files and finding patterns of extra-terrestrial life in radio telescope data. These case studies are interesting for two reasons. First, Pastiche can be viewed as a precursor to a proliferation of file backup and sharing cloud applications, where users pay for storing their files with certain guarantees of fault tolerance and data recovery. Second, Pastiche shows

how difficult it is to obtain guarantees of security and fast access to data in P2P computing. Finally, with SETI@Home, the idea of obtaining slices of CPU and memory resources of the other participants of the network is realized in the P2P setting.

SETI@Home was a radical departure from the algorithm-centric computing, where a small amount of data is processed by complex algorithms to data-centric computing, where a large amount of data is split across multiple processing nodes to run some algorithm over the data in parallel. In both cases, we see a radical departure both from monolithic programming and from a theoretical model of diffusing computation. The differences with monolithic programming are obvious, since Pastiche and SETI@Home are implemented using distributed objects that communicate by exchanges messages using the Internet. When it comes to the basic distributed computing models, we see that these approaches impose additional constraints related to allocation of resources (i.e., storage space and CPU/RAM) on demand based on their availability to distributed objects. In addition, we see that these approaches deal with very large data set with complex structures, often referred to as *big data*. Processing big data requires clever allocation of resources and high levels of parallelism.

2.2.1 Backing up Files With Pastiche

An idea of backing up files is as old as commercial computing. Data are stored on electronic devices (e.g., hard drives) and these devices may break and lose the data. A simple idea of data backup is to replicate the original data on many independent storage devices, thus reducing the probability that this data can be lost, since it is highly unlikely that all these storage devices can break at the same time. This is a simple and a powerful idea that has a major drawback – this cost of these storage devices is somewhat high. Not only does the cost include the price of a storage device, but also its maintenance and support, since a storage device should occupy some space in a ventilated area (hence electrical and room fees) as well as fixing or replacing broken devices. Given that companies have petabytes of data, backing it up cost-effectively is a big and important problem.

A simple idea behind Pastiche is to use a P2P network for backing up peer data [39]. Each peer runs a Pastiche process on his/her computer that connects to the P2P network, determine which chunk of local data to back up, selects peer computers as destinations for these chunks of data, encrypts the data, and delivers them to the destination peer computers for storage. The solution seems straightforward, however, there are questions that make it difficult to implement Pastiche.

- What happens when peer computers that store backed up data become unavailable? Recall that there is no centralized control in P2P networks, so its computing nodes come and leave at will. Not being able to obtain backed up data when the main storage failed renders such backup system unusable.
- Next, how does a peer find other peers to store backup data? How many other peers to choose for the backup?
- Since a P2P network may have tens of thousand of connected peers, how to locate peers that store backup data to retrieve them efficiently without flooding

the network with queries?

- Backup data may contain sensitive information, which could cause serious harm to the individual, organization or company owning it if this information is compromised through alteration, corruption, loss, misuse, or unauthorized disclosure [155] [3, pages 137-156]. A problem is how to distribute chunks of backup data across peer computers in a way that no sensitive information is revealed to the peers who store the data.
- Finally, a question is how to reduce the redundancy of backup data. Since popular operating systems like Linux, Windows, and MacOS are installed on many computers, it is highly likely that many files that users decide to back up may belong to the same OS and other software package distributions. If many users decide to back up common libraries of MS Office, it will result in flooding the P2P network with many backup request messages and wasting storage resources.

As the reader can see, just because an idea is plain and can be stated using a simple language, it does not mean that it can be easily implemented. All raised issues are serious enough to render naive implementations completely unusable.

Question 5: Describe a design where the P2P model is used in combination with a centralized backup facility to address the questions above.

To summarize, these issues are scalability, reliability, availability, and security. *Scalability* refers to the ability of the system to maintain desired performance characteristics by provisioning resources as a function of the (sub)linear complexity of the increased load. In Pastiche, it means as the number of peers in the P2P network increases tenfold, it would take no more than a tenfold increase in resources to support the same time to backup or retrieve data, or without the additional resources the backup time should not grow more than tenfold, and desirably it should not increase at all. If the backup and recovery time increases polynomially or exponentially as the function of the number of added peers, then such system would be deemed non-scalable. A problem with nonscalable systems is that they quickly run out of resources to support increased loads and they stop providing services to their clients.

Consider an all-connected model of a P2P network that is shown in Figure 2.1. Nodes represent peers and edges represent communication channels among peers. This model demonstrates an all connected (i.e., clique) graph, where each peer can communicate with all other peers and the communication channels are bidirectional. Since there are n peers in the model, the number of edges is measured as $n(n-1)$. Assuming that the number of messages is proportional to the number of edges, the overall complexity of the messaging load on this P2P network is $O(n^2)$. That is, suppose that 100 peers join a P2P network and start sending queries to one another and other peers who respond to them. Now, we are talking about an order of tens of thousands of messages that can easily saturate the network and render it nonresponsive. A scalable solution would limit the communication overhead to keep the number of messages (sub)linear in the order of the number of peers who join the network.

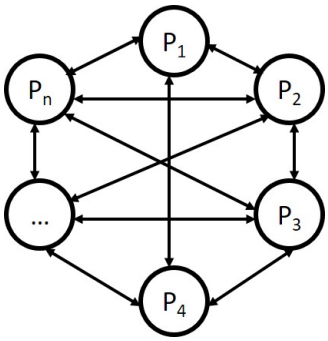

Figure 2.1: A clique of P2P network.

Availability specifies the proportion of the operational time within specified constraints for the system w.r.t. some time interval. For example, a Pastiche implementation may specify that it stores and retrieves backups less than 1Gb within 500ms with 99.999% availability, also known as *five nines*. It means, that given that a non-leap year contains $365 \times 24 \times 60 \times 60 = 31,536,000$ seconds, the system will not be available w.r.t. the specified constraints 0.001% or 315.36 seconds a year or a little over of five minutes. When removing one nine from the five nines, making it four nines availability guarantee, the downtime increases to approximately one hour per year. A small difference between 99.999% and 99.998% doubles the downtime to approximately 10 minutes per year. Ultimately, a goal of every system is to be scalable and highly available, however, it comes at a cost of having to create a clever design and to determine a trade-off among features.

Moreover, a P2P backup system should be *reliable*, i.e., there is some small probability of system failure within some specified period of time. A popular measure, *mean time to failure (MTTF)* specifies an average period of time within which a system functions without failures. Even if there is a failure of some components, this failure can be masked automatically, so that the system continues to be available to process requests from clients. In Pastiche, if a peer that holds backup data fails, the same data can be available from some other peer, i.e., a peer failure is managed through redundancy, which is a trade-off between waste of resources and the reliability of the system.

Question 6: Can a reliable system be non-scalable? Can a highly available system be non-reliable? Can a scalable system be unavailable?

Pastiche answers these questions by resting on three enabling technologies that we discuss later in detail in this book. The first is the P2P routing infrastructure where each node "knows" a few other nodes located in its proximity. That is, a special overlay network is automatically created on top of the P2P network, where peer nodes query a limited subset of all nodes to discover other nodes in a proximity that is calculated using a distance metric based on the content of files.

To deal with redundancy, Pastiche uses content-based indexing using a fingerprinting algorithm, which computes a small (and frequently) unique bit strings called *hashes* for large data sets (i.e., files). In general, hash functions map bits of data of a bigger size to bits of data of a smaller size called hashes. Depending on hash functions, hash functions for two different data sets may reflect similarities between these data sets, or they may be completely different. If hash values are the same for two data sets, it may be the case that these data sets are the same, which in our case may indicate that backing them up would be redundant and only one data set will be backed up; or data sets may be different, and the reason that their hashes are the same is that it is a result of

collision. Many hash functions provide theoretical guarantees that collisions are very rare events.

Once fingerprints for files are computed, lower bits of these fingerprint values are used to determine offsets in the data to break it into chunks. Before these chunks are distributed across some peers in the overlay network, these chunks are encrypted using *convergent encryption*, where encryption keys are derived from the content of these data chunks using hash functions. Convergent encryption gained popularity in cloud computing, since it enables automatic detection and removal of duplicate data while maintaining data confidentiality.

Question 7: Explain if a confirmation of a file attack is possible in Pastiche.

An overlay network in Pastiche is organized based on hashing each computer's fully qualified domain name. For computers that are located in the same domain, the distance between hash values will be closer when compared to computers that are located in different domains. The reader already notices how hashing works with other technologies to address issues of backup. In a way, it is used for *load balancing* to distribute workloads across multiple computing resources, or in the case of backup, to distribute chunks of data equally among different peers. Pastiche is emblematic in how different ideas are brought together to take a monolithic application of backup and make it a distributed system that addresses multiple conflicting objectives using a clever combination of ideas from different fields of computer science.

2.2.2 <Project Name Here>@Home

In the second half of 1990s, personal computers became pervasive and many of them were connected to the Internet. These computers grew powerful and became capable of executing complex algorithms using a fraction of resources that were required in 1980s. However, the resources of the most of these computers were under-utilized. At large companies, tens of thousands of computers remained turned on at night running only screen savers. At the same time, due to the rapidly decreasing cost of computer storage, large amounts of data were digitized, stored on hard drives, and made available for further processing. What we see as a precursor for finding patterns in big data, various projects were started whose names were followed by the `@Home`. A main idea of these projects was to utilize idle personal computers to apply data mining algorithms to search for patterns in large collections of data sets.

For example, SETI@Home project stands for *Search for Extra-Terrestrial Intelligence (SETI)*. The Arecibo observatory is located in Puerto Rico and it hosts a radio telescope whose 1,000 feet in diameter main collecting dish is largest among existing radio telescopes. The telescope dish receives radio signals from the outer space, and it is thought that these signals may contain encoded information that extra terrestrial intelligence, if it exists, sends with these signals. These signals are obtained and coded into data sets, approximately one terabyte of data for one day of observation. Since many of these signals are independent from one another, the data sets can be processed

independently to determine if they contain patterns. Since processing requires a significant amount of computing power, an idea was to enlist volunteers who own personal computers that are connected to the Internet and these volunteers will install a program that will download data sets, process them, and send results back to the server, hence an instance of diffusing computation in practice.

Of course, a key issue is that the installed program should kick in only when computer resources are idle, otherwise, the users will compete with the installed program for resources, a situation that the users will likely resolve by uninstalling the program. To prevent this situation from happening, a clever solution was created to design the program as a *screen saver*, which is a special type of application that are triggered by the operating system when the mouse and the keyboard have been idle for a specified period of time. Originally, screen saver applications had two main goals: to protect a screen from deteriorating, since projecting static images for a long time burns phosphor at certain locations of the screen more than at the other locations and to hide sensitive information. In SETI@Home, a screen saver application was cleverly chosen to delegate monitoring of idleness to the operating system, which will trigger this application automatically. The application will connect to a SETI server, which will send data to the application for processing. In the extreme event when a computer dies, the SETI server will resend this data to some other computer for processing, if a response from the dead computer was not received within some predefined period of time.

Question 8: Describe the implementation of a program that runs in the background that determines the utilization of the CPU to create threads that can participate in P2P tasks without affecting the performance of the workstation.

SETI@Home is not the only project that uses this idea to distribute computations to computers owned by volunteers. Other projects named Milkyway@Home, Folding@Home, and Einstein@Home also use this computing model with a common theme in which data items in data sets are largely independent from one another and these data sets can be easily split and distributed among different computing nodes. Like Pastiche, available resources of computing nodes are borrowed to achieve some computing goal.

In the beginning of 2020 the entire world was ravaged by the Covid virus epidemics that led to the death of tens of thousand of people and many government instituted an almost complete shutdown of many economic activities, so that people stayed home and did not spread the virus. As part of finding a vaccine or a medication to cure the ill, the virus was studied and the biological interactions between various proteins were modeled. In general, computer simulations of biological systems are highly parallel, since there is a need to explore many different developments of the same system for different input parameter configurations in parallel.

To assist with finding the cure faster, the project Folding@home was created to use distributed computing power to simulate the dynamics of proteins in the covid virus. The European Organization for Nuclear Research (CERN) contributed 10,000 computer cores to Folding@home that at some point ran over 230,000 processor cores using 15,000 servers. Users installed a client program from the website of Folding@home and this program pulled a portion of the covid simulation data to run it in parallel on

the users' computers. It is just one example of how harnessing distributed computing makes our lives better.

However, @Home applications operate on big data that does not have to be stored across multiple nodes. On contrary, this big data is mapped to computing nodes and processed by algorithms that are run on these computing nodes, which reduce the input data to much smaller results that are sent to other nodes for further processing.

2.3 Grid Computing

Grid computing takes the ideas of @Home projects and P2P networks to expand it to the idea of a virtual computer, where each user of this computer has a uniform view of the computing platform that hides the complexity of a large number of distributed heterogeneous (i.e., having different structures and interfaces) autonomous computing devices that are connected by a computer network. A main goal of grid computing is to unify these devices to solve a problem, after which these devices can abandon the grid or will be repurposed to solve some other problem. Whereas a user is presented a view of a single computer, this view is virtual, since the user's computer runs a grid computing application that simulates and supports this view using resources from computing devices from volunteers that connect to this grid computing application. Applications of grid computing are computationally intensive and they process large amounts of data to determine certain patterns, for example, genome discovery, weather forecasting, and business decision support.

Question 9: Can the virtual computer be viewed as a realization of an abstraction of the central coordinating computer for a P2P network?

Grid computing is a combination of shared computing (also known as CPU scavenging), where under-utilized computing resources are organized in a grid, and *service-oriented computing*, where computing resources from different ownership domains are organized and utilized to facilitate some distributed computing task. To support grid computing, the Globus Toolkit, a collection of programs and libraries, was developed in late 1990s to enable a community of users and developers to collaborate on developing and using open source software and its documentation for resource federation in virtual organizations for distributed computing [52]. Each participant installs a client program on her computer that connects to the Globus server to obtain access to federated resources, i.e., ones that exist on computers that are distributed w.r.t. a specific client and these resources cannot be easily replicated or the client's computer. The kernel of the Globus server is the *Grid Resource Allocation and Management (GRAM)* service, whose job is to acquire access to client computers, configure them for specific tasks, and initiate, control, and monitor execution of a program that accomplishes these tasks. In addition to GRAM, Globus provides file transfer, indexing, and other capabilities that are important for clients to handle distributed tasks.

2.4 All Roads Lead to Cloud Computing

The use of P2P networks and grid computing enables users to tackle many specialized computationally intensive tasks ranging from DNA sequencing to SETI. Whereas these tasks attract headlines in mass media, more mundane tasks, such as payroll and taxes also require a significant amount of computing power. Companies and organizations accumulate big data and they are eager to extract patterns from this data to improve their decision making processes. As part of their annual capital expenditures, companies and organization routinely purchased many computers and put them in their datacenters, where they host company's data and run company's software. The cost of creating and maintaining datacenters is very high, and since computers become obsolete within a couple of years, the capital expenditures on purchasing new computers is a serious drain of resources that weigh heavily on companies and organizations.

One problem with grid computing and using P2P networks for various business computing tasks is that the needed computing power is never guaranteed. Computers can enter and exit the network at will, and the availability of computing power is difficult to guarantee. Distributing sensitive data to computers that are not owned or controlled by companies may violate data confidentiality policies. Finally, allocating enough resources for a computational tasks is a problem in the context where tens of thousands of tasks compete for resources. Since computing resources are limited, balancing these resources among different tasks is a big and important problem.

Several trends emerged in the middle of 2000s. Companies and organizations started looking for outsourcing the business of creating and owning datacenters. At the same time, prices for computers were dropping precipitously and multicore computers were becoming a commodity in the early 2000s with clock rates increasing to several gigahertz. Large data processing companies like Google, Amazon, and Facebook built data centers to store and process big data and to support millions of customers using web interfaces. That is, each of these companies and organizations presented their abstraction as a distributed object with well-defined interfaces that their customers can access via browsers. As a natural progression, the idea emerged to build commercial datacenters that host tens of thousands of commodity computers. Users can create accounts with companies that own these datacenters and buy computing services at predefined rates. To the users, the datacenter looks like a big cloud that contains computing resources. The users deploy their software applications and send requests to clouds, which process these requests and send response messages. Hence, cloud computing was born.

A few definitions of cloud computing exist. One definition by a group that includes a founder of grid computing states that cloud computing is a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted virtualized, dynamically scalable, managed computing power, storage, platform, and services are delivered on demand to external customers over the Internet [53]. A popular definition is given by the *National Institute of Standards and Technology (NIST)* that states that cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [97].

```
set the instruction pointer to some instruction address
while( Virtual Processor is running ) {
        Read the address from Program Counter
        Fetch the opcode from the address
        Fetch the opcode operands
        Execute the opcode
        increment the instruction pointer to point to the
        address of the next instruction }
```

Figure 2.2: A pseudocode for a virtual CPU (vCPU).

To understand these definitions, let us analyze their key elements. A large-scale distributed computing paradigm implies the departure from small distributed applications where a handful of clients interact with a single object located in a separate address space. The emphasis is on large-scale, where millions of clients access hundreds of thousands of distributed objects that interact asynchronously and for which specific availability and reliability guarantees are provided with *Service Level Agreements (SLAs)* that include constraints on the protocols between clients and server objects, guarantees, and penalties for violating these guarantees. This paradigm is driven by economies of scale, a concept in economics that with the increase of production of certain units in an enterprise, the production cost of each unit drops. For example, moving from hand-crafting each car to their mass production using automated manufacturing lines reduced the cost of cars drastically. The idea of applying the concept of economies of scale to cloud computing is by aggregating commodity computers in a datacenter and selling slices of computing resources to customers, the cost of solving computing tasks will be drastically reduced, since customers would not have to own this cloud computing infrastructure, they will rent it, and the cost will be amortized across many customers.

Question 10: Discuss how resource (de)provisioning can affect the functional correctness of application.

The other parts of the definitions refer to as “abstracted virtualized, dynamically scalable, managed computing power, storage, platform, and services” and “a shared pool of configurable computing resources ... that can be rapidly provisioned and released with minimal management effort.” The notion of an abstract virtualized resource is a key to understanding the benefit offered by cloud computing. Opposite to an abstract resource, a concrete resource has specific technical characteristics and parameters. For example, a concrete CPU has a specific instruction set, the clock rate, and many other details on its structure, caches, and so on. However, when talking about an abstract CPU, such details are omitted, because they may be irrelevant. A case in point is an implementation of a CPU in a software loop that is shown in Figure 2.2.

The processor software runs in a loop. At each iteration it reads the address of the next opcode from Program Counter (PC), a register that keeps the address of the next operation code (opcode). Once the CPU obtains the address, it fetches the opcode from the code area at the given address, fetches the operands, and executes the opcode. This

process is repeated until the CPU loop is shut down. A CPU that this loop represents is highly abstract, since no concrete details on the physical organization of the arithmetic and data units are given. In fact, it is not clear what the speed of this CPU. Thus, the CPU is abstract in a sense that it represent the core functionality of a processing unit.

Moreover, the CPU does not exist as a physical hardware unit, however, it is implemented in software and it can be instantiated and shut down at will. We referred to such hardware as *virtual* hardware. It emulates the real hardware design and device drivers. Of course, we need real physical CPUs to execute instructions on a virtual CPU. And here comes a benefit of virtualization: virtual resources can be instantiated on demand and assigned to physical resources dynamically to scale executions. Users of the cloud computing will not know and will not care how many physical resources are assigned to their applications, where they are located, and how utilized these resources are; all they care is that enough computing power is provisioned to their applications if they need more computing power to maintain the SLA. The shared pool of physical computing resources is managed by the cloud platform, which assigns resources from this pool to specific applications. For example, a webstore application may experience peak loads where additional CPU and RAM should be provisioned to the application rapidly to maintain a desired response time, and then when the user load decreases, these resources will be de-provisioned.

Question 11: Discuss how software bugs in the implementations of the virtual CPU can affect the correctness of cloud-based applications that run on this vCPU.

Recall that a key element of cloud computing is that users pay for provisioned resources. Unlike grid computing, users of cloud computing enter payment information when they create their accounts. Once the account is created and a payment source is verified, all services that the user request from the cloud are reflected in a bill that applies charges for provisioned resources. In that, cloud computing makes a significant departure not only from grid computing and other forms of P2P computing, but also from a custom-built company-owned datacenter. Once the capital expenditure is made in the latter, there is little need to dynamically (de)provision resources, since these resources are already paid for and applications grab as many resources as possible, even if they do not fully utilize them. However, having to pay for underutilized resources in the cloud changes the nature of application deployment, since the cost of computing may be so high that it may not be cost-effective to move applications to the cloud. Thus, it is important to take into accounts how to engineer applications from distributed objects, so that its deployment in the cloud would be cost-effective. This is a major topic that this book is about.

2.5 Summary

In this chapter, we trace the path from early ideas of distributed computing to cloud computing. We start off by describing the abstract model of diffusing computation where computing units are organized in a tree, then we move on to peer-2-peer and grid

computing implementations that was introduced decades later. Along the discussion we introduce key milestones and seminal ideas and concepts that we will use throughout this book. Even though peer-2-peer and grid computing with various @Home projects require a separate book, we extract only most representative ideas that are essential to understand how we arrived to cloud computing. We conclude with a short discussion on virtualization and cloud datacenters to give the readers a convergent view of cloud computing as a natural progression of early ideas and implementations of research in distributed computing.

Chapter 3

Cloud Models and Operational Environments

Large complex software applications require sophisticated hardware and software infrastructures in order to deliver high quality services to their users. As we already discussed, historically, many organizations have chosen to own, manage, and operate their infrastructures, bearing the responsibility for guaranteeing that they are sufficient and available. As a result, such organizations must periodically evaluate their infrastructures and forecast their resource needs (i.e., CPUs, memory, storage). Armed with these forecasts they then purchase and maintain the resources required to meet those needs. Since it is difficult, if not impossible, to exactly predict resource needs, especially over a long period of time, these organizations often end up grossly over-provisioning or grossly under-provisioning their infrastructures. And therefore, they have found it difficult to cost-effectively provide for their long term computing needs. In this chapter, we describe a model of cloud computing and how it is realized using different operational environments.

3.1 A Model Of Cloud Computing

Cloud computing is a system service model in which stakeholders deploy and run their software applications on a sophisticated infrastructure that is owned and managed by third-party providers (e.g., Google and Amazon AWS). Two fundamental properties of cloud computing platforms include provisioning resources to software applications on demand and charging the owners of these applications for pay-as-you-go resource usage. Many cloud providers claim that their cloud infrastructures are *elastic*, i.e., they automatically (de/re)allocate resources, both to *scale out* and *up* – adding resources as demand increases – and to *scale in* and *down* – releasing resources as demand decreases. Highly elastic clouds can rapidly, cost-effectively, and automatically re-provision resources in response to changing demand, and such elasticity can make it seem as if the cloud offers virtually unlimited resources, while expending no more resources than necessary [117].

Figure 3.1 depicts our model of cloud deployed applications. In this model software components run inside *virtual machines (VMs)*, which provide simulated computer environment by emulating physical resources [122, 143]. These VMs are depicted as rectangles and the software components running inside them are depicted by fat black dots. CPU resources are designated with hexagons labeled with the letter P and memory resources are indicated by triangles. In practice, an application can comprise multiple VMs and its software components can be run in these VMs for a variety of legal, geographical, and performance reasons. The assignment of incoming requests from clients to VMs is handled by the *load balancer (LB)*, a component whose goal is to distribute requests for computing services (i.e., *workloads*) to optimize the overall performance of the system using multiple criteria: to achieve equal resource utilization, to minimize response time or to maximize the number of serviced requests per some time interval, and to increase availability of the system by routing requests to resources that can service them if other resources are not available.

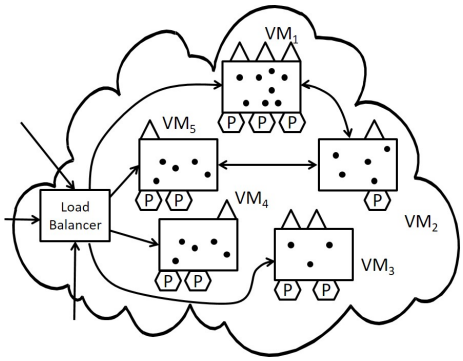

Figure 3.1: A model of the cloud.

Question 1: Explain how Pastiche-related data hashing can be used for load balancing in the cloud.

In our model, physical hardware, the LB, VMs and communication interfaces comprise the key elements of the cloud computing platform and are the key means through which service costs and resource allocation are controlled and manipulated. Client requests (shown with incoming arrows in Figure 3.1) arrive to the LB that then distributes these client requests to specific VMs. Some VMs (e.g., $\mathtt{VM}_2$) may not receive requests directly from the load balancer but from other VMs. Resources are allocated to VMs to speed up the execution of the components that process incoming requests. A simplified sequence of steps for cloud processing is the following.

1. LB receives a set of requests $\{R\}$ from different clients of the hosted software.
2. LB forwards different subsets of $\{R\}$ to VMs.
3. VMs receive $\{R\}$, methods of software are invoked.
4. Cloud infrastructure collects performance counters and trigger client-defined scripts that perform resource-provisioning actions.

In general, the cloud allocates different amounts of resources to different VMs as indicated in Figure 3.1. Clouds use several common strategies for doing this, including

scaling up and scaling out. For example, since $\texttt{VM}_1$ contains more components that involve computationally intensive operations, it is scaled up by assigning three CPUs and memory units. Alternatively, the cloud could scale out this application by replicating VMs (e.g., $\texttt{VM}_4$ and $\texttt{VM}_5$), thus enabling multiple requests to be processed in parallel. That is, the cloud uses two main scaling operators: $\texttt{sres}(r,a,i)$ and $\texttt{sinst}(a,i)$, where r is the type of a resource (e.g., memory, CPU), a is the amount, and i is the VM identifier. The scaling operator `sres` (de)allocates the resource, r, in the amount a to the VM, i, and the scaling operator `sinst` (de)allocates a instances of the VM, i. In theory, elastic clouds "know" when to apply these operators to (de)allocate resources with high precision.

Question 2: Is it possible to create a general algorithm that can predict statically when an application will terminate in its execution for an arbitrary set of input values? Can such an algorithm be used to apply scaling operators effectively to deprovision resources right before the application will terminate?

Unfortunately, the gap between the theory and practice is large. Consider a situation when an application load increases, e.g., numerous customers flock to a web-store application to buy a newly released electronic gadget. Once the LB receives an increased number of request in step `1`, the cloud should provision resources to VMs in anticipation of the increased load, so that by the time that the LB forwards client requests in step `2`, the VMs are ready to process them in step `3`. In general, the cloud "does not know" what scaling operator to apply and what parameters to use for these operators. It is only in step `4` after analyzing performance counters the cloud enables stakeholders to apply scaling operators. The delay between the steps `2` and `4` leads to over- and under-provisioning, which is typical for existing cloud infrastructures [62]. Naturally, if the cloud "knew" the time and extent to which resource demands would change then it could precisely and proactively (de)allocate resources, thereby improving its elasticity and resulting in $\Delta_r \to 0, \Sigma_g \to 0$.

The attraction of elastic clouds is that stakeholders pay only for what they use, when they use it, rather than paying up-front and continuing costs to own the hardware/software infrastructures and to employ the technical staff that supports them [13, 23, 24, 93, 98, 123]. Of course, in practice, even the most elastic clouds are not perfectly elastic [77]. Understanding when and how to reallocate resources is a thorny problem. Allocating those resources takes time, and it is generally impossible to quickly and accurately match resources to applications' needs over extended periods of time. An article by the Google Cloud team underscores this point as it describes a state of the art supervisory system that lets applications monitor various black box metrics and then direct the cloud to initiate scaling operations based on that data [62]. However, the very existence of the system confirms the large gap between the promise and the reality of elastic cloud computing. For example, a documented limitation of this system is that its polling approach substantially lags changes in resource usage, rather than proactively anticipating changes. As a result, main problems of cloud computing elasticity is either *under-provisioning* applications where they lack the resources to provide appropriate quality of service, or *over-provisioning* where the stakeholders would be holding and

paying for excess resources.

3.2 Cost And Utilization in Cloud

Figure 3.2 depicts a notional a cost-utilization graph, in which application's usage of resources is represented by the thin (red) line and the allocation of resources by the cloud infrastructure is represented by a thick (blue) line. Assume that at time A_1, application load increases, e.g., numerous customers flock to a web-store application to buy a newly released electronic gadget. Therefore, the cloud allocates extra resources to this application to match its increased load. Ideally, the resources allocated, the thick (blue) curve, should subsequently match resources used, the thin (red) curve. Unfortunately, this is often not the case for modern cloud providers. The divergence between resources needed and resources allocated is indicated by peaks of over-provisioning at times O, and by valleys of under-provisioning at the time U in Figure 3.2. The application owners overpay at the times O for resources they do not need, while their applications cannot provide high quality services at the time U. This situation holds for de-allocation as well and it is typical for existing cloud infrastructures, such as the Google cloud [62]. Naturally, if the cloud "knew" the time and extent to which resource demands would change then it could precisely and proactively (de)allocate resources to meet that demand, thereby improving its elasticity.

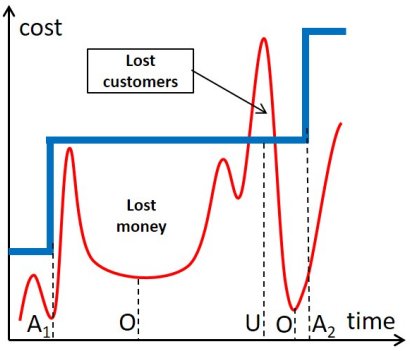

Figure 3.2: An example of the cost-utilization curve, where actual application usage of resources is shown with the thin (red) line and the allocation of resources by the cloud is shown using the thick (blue) line. Modified from source: amazon.com/ec2.

> **Question 3**: What information is needed in advance to schedule the execution of applications in the cloud most optimally?

Two fundamental elasticity measures include the 1) interval of time, Δ_r, between the moment that an application needs resources to change and the moment that the cloud changes these resources, and 2) the charge granularity, Σ_g, that the cloud platform uses to compute the cost of the granted resources [77]. The higher Δ_r and Σ_g are, the less elastic the cloud infrastructure is and consequently, the greater the gap between the resources needed and the resources allocated.

Although elasticity is a fundamental enabler of cost-effective cloud computing, existing *provisioning strategies* (i.e., rules that (de) allocate resources to applications) are obtained ad-hoc by programmers who study the behavior of the application in the

cloud. It is a manual, intellectually intensive and laborious effort, and these provisioning strategies are often not as effective as they need to be. It is a symptom of a bigger problem where software engineering is viewed orthogonal to cloud computing – with current approaches, stakeholders design and build software applications as if a fixed, even if sometimes abundant, set of resources were always available. Since no software engineering models are used to deploy applications in the cloud, resources are frequently either over or underprovisioned, thereby increasing the applications' cost and degrading their quality of service. With global spending on public cloud services estimated to reach $110.3Bil [12, 58] by 2016, and with an estimated $20Bil of US federal government spending on cloud computing [84], *the cost of not having truly elastic clouds will be measured in the billions of dollars* [14].

3.3 The State of the Art And Practice

Today, stakeholders typically deploy their applications to the cloud, using ad-hoc scripts in which they encode the behavior of these applications and how the cloud should apply the scaling operators based on a coarse collection of the performance counters in step 4 and otherwise "guesstimating" on how to provision resources to their applications in the cloud. The Google Cloud, the Amazon EC2 and Microsoft Azure clouds have issued guidelines for manually load balancing their elastic clouds, directing their users to manually specifying the conditions that trigger work routing and scaling operations. The Amazon EC2's documentation on auto scaling, for instance, states that "use the ... Amazon CloudWatch command to create an alarm for each condition under which you want to add or remove Amazon EC2 instances, and specify the Auto Scaling Policy that you want the alarm to execute when that condition is met. You can define alarms based on any metric that Amazon CloudWatch collects. Examples of metrics on which you can set conditions include average CPU utilization, network activity or disk utilization."

The key take away here is that existing cloud providers understand that their users often need to manually configure the cloud. Such manual activities are tedious, error-prone and expensive, and clearly demonstrate that clouds, such as Amazon's EC2, have a long way to go in their quest for better, more automated, elasticity (see `http://aws.amazon.com/autoscaling/`).

> **Question 4**: Discuss ways to improve autoscaling to optimize resource provisioning in AWS.

Most current software development approaches pre-date the cloud, and as a result they effectively treat cloud applications no differently than traditional distributed systems. If anything, some might argue that separating the software system from the cloud execution platform should allow engineers to worry much less about performance, as it will, after all, be boosted by the cloud. While it is true that developing software for public cloud computing differs little from developing for in-house infrastructures, there are at least two fundamental differences between these environments. First, cloud comput-

ing depends on *time-varying resource allocation* and, second, it apportions *costs based on runtime usage*. In fact, these features are necessary to enable and manage cloud elasticity [117]. In addition, cloud providers rely on slow, reactive, black-box approaches to dynamically provision resources, which leads to higher cost and degraded performance [29, 33, 68, 110, 140].

Thus, there is a problematic disconnect between existing software engineering approaches and the needs of applications that are deployed in the cloud. With current approaches, stakeholders design and build software applications as if a fixed and abundant set of resources would always be available. This, however, is simply not true for cloud computing. *Resources are dynamically allocated and deallocated to different components, which has important cost and performance implications that can even violate assumptions made in requirements specifications.*

Developers need such support throughout the software lifecycle to answer questions, such as: how to assign features to components and then how to assign resources to those components in ways that reduce overall cost and maximize performance; how to ensure that performance problems are found during performance testing, so that these problems will not result in the excessive costs when the application runs in the cloud; how to ensure functional correctness when scaling out the application automatically; and how to ensure that scaling strategies are appropriate for a given application.

Providing support to software engineers who create, deploy, and maintain software applications for the cloud requires the creation of a paradigm that addresses multiple cloud-relevant engineering concerns throughout the software engineering lifecycle. For instance, both the engineering approaches that developers use to create applications a nd the cloud computing infrastructures that ultimately run these applications must take resource variability and cost models into account. For example, with usage-based cost models, system performance problems should map directly to quantifiable economic losses. These considerations can be used both to reason about and direct a system's up-front design, and to direct resource provisioning.

3.4 Measuring Cloud Elasticity

In general, `if-then` elasticity rules contain antecedents that describe the level of utilization of some resources (e.g., CPU utilization $\geqslant 80\%$) and the consequents that specify (de)provisioning actions (e.g., (de)provision a VM). Unfortunately, rule creation is an error-prone manual activity, and provisioning certain resources using manually created rules may not improve the applications performance significantly. For example, when the CPU utilization reaches some threshold due to a lot of swapping or a lack of the storage, provisioning more CPUs does not fix the underlying cause that requires giving more storage to the application. That is, often rules are not optimal in terms of allocating required resources based on projected applications needs [22]. Unfortunately, customers often pay for resources that are provisioned to, but not fully used by their applications [77], and as a result, the benefits of cloud computing may be significantly reduced or even completely obliterated [7]. This is a fundamental problem of cloud computing and it is of growing significance, since it affects the technology spending in the excess of $1 trillion by 2020 [152].

Question 5: What is a relation between the terms elasticity and scalability? Can an elastic computing environment be non-reliable?

It is very difficult to create rules that provision resources optimally to maximize the performance of the application while minimizing the cost of its deployment. Doing so requires the applications owners to understand which resources to (de)provision at what points in execution, how the cost of the provisioned resources varies, and how to make trade-offs between the applications performance and these costs. It is difficult to determine what resources to allocate at runtime, even for five basic resource types (i.e., CPU, RAM, storage, VM, and network connections) where each type has many different attributes (e.g., the Microsoft Azure documentation mentions 30 attributes [104,105], which result in tens of millions of combinations). Furthermore, suppose that the performance of an application falls below some desired level that is specified by the application's owners. Since there are multiple possible combinations of resources that could be allocated to the application, the challenge is to find the rules that provision only minimally needed resources to maintain the desired level of performance (i.e., the average response time). Conversely, provisioning resources that are not optimal often leads to a loss in customers revenues.

Workloads' fluctuations and burstiness reduce the effectiveness of the elasticity rules. Indeed, a rule that is triggered to provision resources for one workload becomes inapplicable a short moment later after the rapid workload change. In our illustrative example in Figure 3.3, a rapid change from the workload 2 to workload 3 results in a situation where the cloud allocates resources according to the rule based on workload 2, whereas different resources are needed to maintain a desired level of performance for workload 3, a short moment after the provisioning is made for workload 2. Finding workloads that lead to the CUVE is very important, where the SLA is violated and the cost of deployment is high because of the provisioned resources. Once known, these workloads and rules can be reviewed by developers and performance engineers who optimize the rule set to achieve a better performance of the corresponding application.

When the cloud infrastructure provisions resources, there is a delay between the moment when the cloud assigns a resource to an application and the moment when this application takes control of this resource. There are at least a couple of reasons for this delay: the startup time for a VM that hosts the application or its components includes the VMs loading and initialization time by the underlying infrastructure; assigning a new CPU to the existing VM requires its hosted operating system to recognize this CPU. In order for a running application to use newly provisioned resources, the host OS must load their drivers and schedule the instructions of the already running applications to access and manipulate these resources [95], which takes from seconds to tens of minutes [146]. Interestingly, the cloud infrastructure starts charging the customer for the resources at the moment it provisions them rather than when the application can control these resources [77]. A similar explanation goes for the deprovisioning of resources. If the application changes its expected runtime behavior during a resource initialization time, this resource may not be needed any more by the time it is initialized to maintain the application's performance. As a result, more often than not, customers

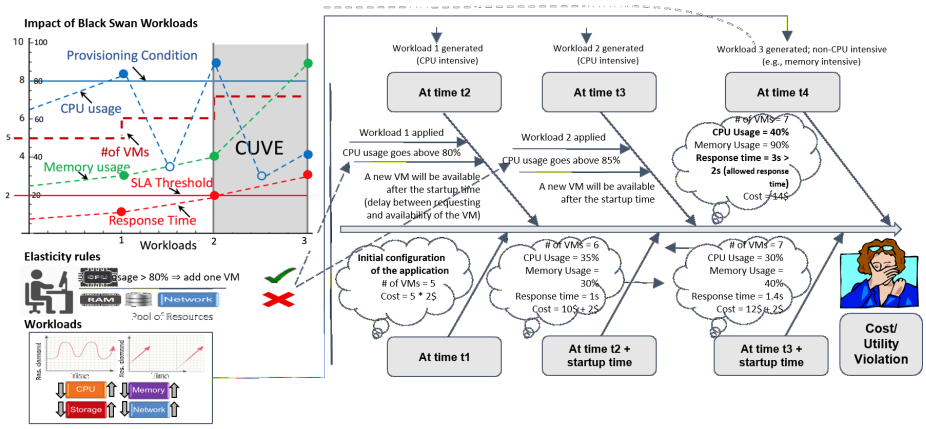

Figure 3.3: An illustrative example of the CUVE for a cloud-based application. The timeline of the operations is shown with the horizontal block arrow in the middle. The process starts with the customer who defines elasticity rules on the left and the events are shown in the fish-bone presentation sequence that lead to the CUVE on the right. (With main contributions from Mssrs.Abdullah Allourani and Md Abu Naser Bikas)

pay for resources that are not used by their applications for some period of time.

Creating an automatic approach for generating rules that (de)provision resources optimally based on the applications behavior is an undecidable problem because it is impossible to determine in advance how an application will use available resources unless its executions are analyzed with all combinations of input values, which is often a huge effort. Currently, many rules are created manually to approximate a very small subset of the applications behavior, and clouds often (de)provision resources inefficiently in general, thus resulting in the loss of customers time and money [71]. Automatically creating test input workloads is a big problem that detect situations when customers pay for resources that are not fully used by their applications while at the same time, some performance characteristics of these applications are not met, i.e., the *Cost-Utility Violations of Elasticity (CUVE)*.

> **Question 6**: Can CUVE be a problem for short-running applications that are executed frequently?

The CUVE problem with a cloud-deployed application is illustrated in Figure 3.3. On the left side, the input is a set of rules for elastic resource provisioning and workloads for the application. The top leftmost embedded graph that is shown in Figure 3.3 summarizes how workloads' fluctuations and burstiness reduce the effectiveness of the elasticity rules. The horizontal axis shows numbered workloads, and the inner measurements on the vertical axis indicate the utilization of CPU and memory in percents. The solid blue line shows the provisioning condition that describes the level of CPU

utilization. The outer measurements on the vertical axis indicate the number of the provisioned VMs and the response time in seconds, and the solid red line shows the threshold of a *service level agreement (SLA)* that indicates a desired performance level (i.e., the response time).

We show that a rapid change from the workload 2 to the workload 3 results in a situation where the cloud allocates resources according to the rule based on workload 2, whereas different resources are needed to maintain a desired level of performance for workload 3, a short moment after the provisioning is made for workload 2. Finding such black swan workloads that lead to the CUVE is very important during stress testing, where the SLA is violated and the cost of deployment is high because of the provisioned resources. The cost and the performance move in opposite directions. Once known, these black swan workloads and rules can be reviewed by developers and performance engineers, who optimize the rules to achieve a better performance of the corresponding application. We show how the interactions between workloads and rules lead to the CUVE problem.

Consider what happens in the illustrative example with the commonly recommended rule that specifies that the cloud infrastructure should allocate one more VM if the utilization of the CPUs in already provisioned VMs exceeds 80%. As an example, we choose the initial configuration of five VMs at the cost of $2 at the time t_1. We rounded off the cost for the ease of calculations and based it on the pricing of various cloud computing platforms [9, 63, 103]. Then, a CPU-intensive workload triggers the rule at the time t_2. A new VM will be provisioned after some startup time while the owner of this application is charged an additional $2 at the time t_2. The VM will become available to the application at $t_2 + t_{VM_s}$, where t_{VM_s} is the VM startup time. Suppose that allocating one more VM in this example decreases the CPU utilization to 35% whereas the memory utilization remains the same at 30%. The new workload 2 leads to a significantly increased CPU utilization, and another VM is allocated at the time t_3. This is in a nutshell how an elastic cloud works.

Question 7: Can CUVE be alleviated by knowing the behavior of the application in advance and using this knowledge to change elastic rules?

Suppose that the response time for the application should be kept under two seconds according to the SLA that is specified by the applications' owners, and a goal of the elastic rules is to provision resources to the application to maintain the SLA. The SLA is maintained below the threshold until the time t_4 when the workload rapidly changes. The new workload 3 leads to a significant burst in the memory usage whereas the utilization of the CPUs in already provisioned VMs remains low at 40%. The memory utilization increases to 90%, and there is no rule that can be triggered in response, thus, subsequently, there is no action taken by the cloud to alleviate this problem. The CPUs wait for data to be swapped in and out of memory, and they spend less time executing the instructions of the application. As a result, the application's response time increases, thus eventually breaking the SLA threshold. Furthermore, at the 40% higher cost, the SLA is violated and the performance of the application worsened significantly, while the application's owner pays for resources that are under-utilized.

An approach for measuring cloud elasticity involves the following steps: 1) submit a workload that follows some pattern; 2) measure the demand of the workload on the cloud platform; 3) measure the supply of the available resources by the platform; 4) measure the latency and some other performance aspects; 5) calculate penalties for under- and overprovisioning of the resources and add the cumulative penalties [77].

3.5 Cloud Service and Deployment Models

In this section, we discuss main cloud service and deployment models.

3.5.1 Software Stacks

Software has become highly componentized and modularized, where software libraries are organized in packages that are logically related to serve some common goals. For example, a user can select an operating system from a dozen of choices, install it in a VM host software (e.g., VmWare, Xen, or VirtualBox), choose a web server, a database to host data that can be accessed by objected hosted in this web server, and decide what protocol to use to host objects that will provide services to clients who will use this protocol. Interestingly, one can create a platform for a specific type of software applications simply by stacking software solutions that are created and built by different software developers, companies and organizations.

Question 8: Design and implement a software stack for web page indexing applications that collect information about web pages on the Internet.

There are currently many software stacks and their number multiplies every month. One of the popular software stack that we will discuss in this book is Map/Reduce, where a specialized file system is built for large-scale parallel processing of data using commodity hardware, Google File System (GFS), and a set of specialized libraries. Some popular software stacks include LAMP that consists of the Linux OS, Apache HTTP server, the MySQL relational database, and the PHP programming language; LYME that consists of the Linux OS, Yaws web server, the Mnesia or the CouchDB databases, and the Erlang functional programming language; and OpenStack, which is an open-source cloud environment implementation that runs on Linux. A variant of LAMP that substitutes Windows OS for Linux is called WAMP.

Question 9: Can a software stack be composed from other software stacks?

A common thread for different software stack is hiding some services within the stack and exposing a limited set of interfaces to the external users of these stacks. Naturally, doing so enables users to concentrate on specific tasks without having to manage unrelated components of the stack. More importantly, depending on the organization of

the software stack, the user may not have any access to certain settings of the components of the stack. For example, allowing some users to configure the virtual memory settings of the underlying OS may affect other users who run applications on top of the VM that hosts this OS. Thus, it is important to understand not only what interfaces are exposed by different components of a software stack, but also what types of users can control what resources.

3.5.2 Cloud Deployment Models

In clouds, there are four deployment models: public, private, hybrid, and the community cloud models. In a deployment model, a cloud provider is an organization that owns the physical resources that constitute the cloud environment, and cloud consumers or customers include persons, organizations, or computing entities that use cloud resources to accomplish some tasks. Consumers access cloud resources as clients using certain protocols and API calls that the cloud exposes.

When a cloud provider can only be the same entity as the consumer, then the cloud is called private. For example, a large company can set up its own datacenter and task a separate division within the company to maintain it. Different departments and divisions of this company can use this internal cloud and pay for resources that they use to the division that maintains the cloud. One may feel that the company moves its own money from one pocket to the other, however, each department has its own budget and its managers allocate it based on the business needs. This constraint leads to more thoughtful decisions on using computing resources. Of course, a reasonable question is why not to use an external cloud provider. Doing so may not be even an option, since the company's leadership chooses to build and use its private cloud to host the data of utmost importance to the company's business and to ensure that certain critical tasks are given a higher priority. For example, a large consulting company that builds software for large financial and insurance companies cannot simply upload the confidential data that belong to their customers to a third-party cloud provider, so instead it built its own private cloud datacenter and hosts the customers' data with proper security. In some cases, due to legal and geographical constraints, there is no other option for companies and organizations but to build their private clouds.

Alternatively, public cloud providers allow all customers to access their services who agree to abide by the service agreement that is issued by the cloud providers. No contract may be needed, customers can pay hourly for provisioned resources. It is often not known to customers what physical servers their VMs run on and they do not have access to hardware to control the performance of their applications that run on top of this hardware. Opposite to it, private clouds enable stakeholders to exercise the full control over the underlying hardware, assigning the most powerful computers to mission-critical applications, limiting multi-tenancy to ensure that other applications will not run on this hardware and compete for resources with these mission-critical applications, and fine-tuning hardware and OS settings to ensure the maximal performance. Thus, private clouds offer much higher customizability over public clouds.

Question 10: A dedicated server is a computer that a customer purchases for exclusive use in the public cloud. Can using dedicated computers in public clouds reduce the need for customers to create and use their private clouds?

Finally, and probably most importantly, public clouds make it difficult for companies to comply with legal and government regulations, e.g., Sarbanes Oxley and the Health Insurance Portability and Accountability Act of 1996 (HIIPA). The latter specifies physical and technical safeguards to ensure the privacy and the confidentiality of the patient's electronic protected health information (ePHI). HIIPA specified strict facility access and control, where all users must be authorized to access computers that store data, which is very difficult to accomplish with public clouds. Technical safeguards ensure that only authorized users can access ePHI with unique user identifiers, activity auditing, automatic log off and encryption and decryption. Public clouds make it difficult to balance two goals: multitenancy and strict control of accesses to resources.

Since public and private clouds have benefits and drawbacks and they complement one another to a certain degree, hybrid clouds has become popular, where a company or an organization creates and integrates its private cloud with its account at a public cloud provider. For example, vCloud Connector from vmWare links private and public clouds in a unified view, so that the users can operate a hybrid cloud without dealing separately with particular aspects of constituent private and public clouds. A common scenario is to establish a secure *virtual private network (VPN)* connection between VMs in the public clouds, so that data can be shared across these clouds as if the VMs that are hosted in the public cloud are part of the private cloud. Finally, community clouds are built and controlled by groups of non-competing companies and organizations that share goals and resources. Non-competitiveness is important, since participants of the community cloud should trust one another. For example, an educational community cloud can provide access to learning resources to educate users who want to learn about the state-of-the-art technologies that the companies and organizations create and sell that are the founding members of the community cloud.

3.5.3 Software as a Service (SaaS)

An example of SaaS is a document management system like Google Docs[1] using which customers can create accounts, create, edit, save, and share text documents and spreadsheets. The underlying cloud infrastructure is invisible to customers, since they connect to the document system's portal and use software applications to manage the documents. Thus, customers have access only to the interfaces exposed by the applications that are deployed in the cloud, not to the underlying cloud. A common way to access application's interfaces is via the Internet, using a browser or by sending requests using some protocol.

As an illustration of the concept that customers can use well-defined *Application Programming Interface (API)* calls to interact with Google Spreadsheet as SaaS, we consider Java pseudocode that is shown in Figure 3.4. This pseudocode can be easily

[1] `https://www.google.com/docs/`

```
1  import com.google.gdata.*;
2  import java.io.IOException;
3  import java.net.*;
4  import java.util.*;
5  public class GoogleSpreadsheetClient {
6    public static void main(String[] args)
7        throws AuthenticationException, MalformedURLException,
8        IOException, ServiceException {
9      SpreadsheetService service=new SpreadsheetService("SheetName");
10     service.setOAuthCredentials(
11         new OAuthParameters(), new OAuthHmacSha1Signer());
12     URL url = new URL("https://spreadsheets.google.com/");
13     SpreadsheetFeed feed=service.getFeed(url,SpreadsheetFeed.class);
14     List<SpreadsheetEntry> spreadsheets = feed.getEntries();
15     for (int i = 0; spreadsheets.size() > 0; i++ ) {
16         SpreadsheetEntry spreadsheet = spreadsheets.get(i);
17         System.out.println(spreadsheet.getTitle().getPlainText());
18         WorksheetFeed worksheetFeed = service.getFeed(
19         spreadsheet.getWorksheetFeedUrl(), WorksheetFeed.class);
20         List<WorksheetEntry> worksheets=worksheetFeed.getEntries();
21         WorksheetEntry worksheet = worksheets.get(0);
22         URL cellUrl = worksheet.getCellFeedUrl();
23         CellFeed cellFeed = service.getFeed(cellUrl, CellFeed.class);
24         for (CellEntry cellEntry : cellFeed.getEntries()) {
25                 Cell cell = cellEntry.getCell();
26                 if (cell!=null)System.out.println(cell.getValue());}}}}
```

Figure 3.4: Java-like pseudocode for a client that uses Google SaaS spreadsheet service.

converted into a working Java program (we leave it as a homework exercise). The class GoogleSpreadsheetClient contains the method main that creates an instance of the class SpreadsheetService in line 9. In order to link this service with a specific account, the user must authenticate herself as it is done in lines 10–11 using the class OAuthParameters that implements the open-source web authentication protocol called OAuth. The discussion of how OAuth works is beyond the scope of this chapter, readers can consult the appropriate standards available on the Internet[2].

Once the client authenticates itself with the Google Docs SaaS, the url object is created in line 12 where the client provides a valid URL to the location of a spreadsheet. This object is used as the first parameter in the method call getFeed for the object service to obtain the object feed of the type SpreadsheetFeed in line 13. The purpose of this class is to obtain a list of spreadsheets at this URL in line 14. Then in lines 15–26 the for loop is executed where each spreadsheet entry is obtained by the index in line 16, its title is printed in line 17, and then for the first worksheet of each spreadsheet its list of cells is obtained in lines 24–25 and then their values are printed in line 26. Many other Google Docs SaaS applications have a similar API call structure and the conceptual structure of the clients is similar to this example.

[2]Specifications of the OAuth authorization framework that enables third-party applications to obtain access to HTTP objects is available at this community website `http://oauth.net/`

A key point of this example is to illustrate that the users of SaaS do not have any access to the underlying hardware or the operating system or other low-level components of the application platform. SaaS expose interfaces whose methods enable clients to obtain access to the application objects, create and destroy them and change their states. However, clients do not have a choice of the platform on which SaaS applications are deployed, they are often not even given interfaces to obtain the information about the platform. On the one hand, this is a strength of SaaS, since clients do not have to worry about the underlying platforms (i.e., the stack); on the other hand, clients are limited in their ability to control and administer the entire stack.

Question 11: Argue pros and cons of a SaaS application exposing the underlying hardware or OS interface.

A key characteristic difference between SaaS, PaaS, and IaaS is the level of control that clients can exercise over the server stack. As we can see from the code fragment in Figure 3.4, the client has the access only to application-level objects (e.g., spreadsheets and their properties and values) using well-defined interfaces that the application exposes, and not to the VM and OS objects and definitely not to hardware components. In PaaS and IaaS, the client can access and manipulate objects that are defined in the scopes other than the application-level scope.

NIST introduces two terms that define capabilities of clients in the cloud: control and visibility. Controlling resources means that a client can determine who can access and invoke methods of objects that the client owns in the cloud environment, and having the visibility means that a client can obtain statuses of objects that are controlled by other clients [97]. In SaaS, clients can have the visibility and control over application-level objects. A cloud provider controls the hardware, the operating system settings, and other components of the software stacks installed in the VMs. Of course, when accessing SaaS via browsers, it is important to ensure cross-browser compatibility, so that the users of different browsers can have the same experience working with SaaS applications. Security is always a concern, since using SaaS interfaces may expose security vulnerabilities of the SaaS implementations.

A limited level of control and visibility carries its own benefits. Since SaaS is installed once in the cloud, it does not have to be installed on clients' computers, and there are no installation and distribution costs. Licensing is improved – clients do not need to purchase multiple licenses and engage in complicated license verification protocols when accessing SaaS objects from multiple computers, since the SaaS server handles license verification automatically when clients access SaaS objects. Finally, clients do not have the visibility and control of the underlying hardware, operating systems, and various software packages that are used to host SaaS applications. On the one hand, the inability to control the underlying software and hardware stack may negatively affect the ability of the clients to achieve better performance of the SaaS application; on the other hand the clients do not have to be burdened with the complexity of the underlying layers of software and hardware, which is exactly a main selling point of SaaS.

Engineering SaaS applications is a critical issue, since many clients share the same applications. Client objects must be effectively isolated from one another, since each

client has sensitive data that is processed by the same SaaS possibly on the same physical computers. If a SaaS application has bugs that expose data structures created by one client to other clients, then sensitive data in one application may be inadvertently exposed to other applications that run on the same physical computer. It is needless to say that security and privacy problems will take a long time to resolve and users of SaaS should be aware of potential implications of selecting a specific cloud model.

3.5.4 Platform as a Service (PaaS)

In PaaS, cloud users can create, deploy and fully control their applications, and cloud providers supply API calls that the cloud users incorporate into their applications to access and manipulate resources in the cloud. The cloud infrastructure is responsible for provisioning resources to applications and for providing interfaces with API calls to interact with resources and other applications. For example, Google AppEngine PaaS provides an authentication mechanism using Google Accounts for applications, supports languages for developing applications that include Java, Python, PHP, and Go, and load-balances incoming requests to multiple instances of the provisioned applications[3]. In addition, Google AppEngine PaaS provides a datastore for storing application data, and multiple interfaces to query data using the Internet, to store into and retrieve data from a special distributed in-memory data storage for improving performance of applications, to search data using the Google search engine, and to access logs that are produced as a result of running applications in the cloud. Amazon, Microsoft Azure, vmWare and many other cloud provides enable these and other PaaS services to their customers.

Question 12: Is LAMP software stack a PaaS or a SaaS?

Key differences between SaaS and PaaS are in the availability of programming access to the stack services below the applications and the level of control over the application and the underlying stack. Unlike SaaS, the cloud provider has no control over applications that customers choose to build, deploy, and ultimately control in the cloud. On the other hand, in PaaS, just like in SaaS, the cloud customer has no control over the operating system and the hardware. Therefore, the key transfer of control in PaaS from the cloud provider to its customers is in the application layer and in the programming layers. Choosing the PaaS model over SaaS makes sense when cloud customers want to build their own applications and deploy them in the cloud.

The success of specific PaaS implementations depends upon the availability of a broad range of programming languages, database management software, various tools and frameworks for application development. Some of these tools and frameworks may be optimized for specific operating system/hardware configurations. For example, a cloud provider may enable interfaces for data caching, where cache storages are allocated on the same hardware where the application runs. That is, whereas the cloud customers have no control over hardware and how the cloud provisions it to applications, using cache interfaces in PaaS will tell the cloud to position the application data

[3] `https://cloud.google.com/appengine/`

caches in the close proximity to the CPUs which execute the instructions of the corresponding applications. Cloud providers strive to make their offerings of PaaS attractive to their customers, so that they can develop applications that will deliver services to their clients more effectively and efficiently than other competing cloud providers.

Different PaaS offerings by many cloud providers create a situation that is called customer lock-in (also known as vendor-lock in or proprietary lock-in). Since different cloud providers offer different programming interfaces and software stack services, an application that is developed using AWS PaaS is not likely to run on vmWare PaaS. To avoid this situation, customers can build their applications using general programming languages and standard development libraries, however, they will most likely not be able to take advantages offered by optimized cloud PaaS interface offerings. In addition, since security and privacy are big concerns, many PaaS interfaces use platform-specific authentication and protection mechanisms, and if an application uses these interfaces, it may be locked into the specific PaaS offering from the moment that the application is conceived. Thus, inter-cloud portability and customer lock-in are big problems and there is no solution that can address all aspects of these problems.

3.5.5 Infrastructure as a Service (IaaS)

In IaaS, every layer of the software stack is under the full control of cloud customers. Only hardware is fully controlled by the cloud provider with customers having no control over hardware. The cloud provider also controls the VM monitor (i.e., hypervisor), which is a hardware or a software module that controls the lifecycle of VMs. Cloud customers use API calls that cloud providers expose to create VMs, checkpoint, suspend, and resume them, provision resources to them, and shut them down. The cloud provider has no control over what operating system customers choose, what software stack they install on top of the operating system, and what applications they build and deploy in the cloud. Using IaaS makes sense when cloud customers have specific software stacks that the cloud providers cannot offer as PaaS, or when they want to avoid vendor lock-in, so that they can quickly redeploy their software to a different cloud provider when their service demands are not satisfied.

Consider an example of the IaaS API using the Oracle IaaS documentation [4]. The Oracle cloud manager exposes interfaces that receive *Hypertext Transfer Protocol (HTTP)* requests from cloud customers, performs services to satisfy those requests, and reply with the status using HTTP. An example of a request to allocate a VM is shown in Figure 3.5. The payload message of the HTTP request is given in the *JavaScript Object Notation (JSON)* format, which is one of the most common formats for specifying the structure and the state of data objects[5]. To understand the syntax of JSON, think of simple key-value pair format in the form "`key`" : `value`, where `key` is a string that designates the name of a field of the data object and the `value` is its value. In line 1, the key `based_on` specifies the location of a template for a requested VM. The key `cpu` in line 2 specified an array of two values, where the square brackets designate the array value. The first number 2 specifies the number of the CPUs allocated to the

[4] `https://docs.oracle.com/cd/E24628_01/doc.121/e28814/iaas_api.htm`
[5] `http://www.json.org/`

```
1 "based_on":"/em/cloud/iaas/servicetemplate/vm/uniqueIdentifier",
2 "cpu" : [2,1000],
3 "memory" : "4000",
4 "params":{
5         "server_prefix":"ZONEPOST",
6         "root_password":"welcome1" }
```

Figure 3.5: Example of HTTP POST request to create a VM at a cloud hosted by Oracle.

```
1  "uri" : "/em/cloud/iaas/server/byrequest/1" ,
2  "name" : "VDOSI VM Creation 1345391921407" ,
3  "resource_state" :     {
4          "state" : "INITIATED" ,
5          "messages" :
6          [ {
7                  "text" : "The Request with ID '1' is scheduled",
8                  "date" : "2016-02-24T13:12:31"
9            } ]} ,
10 "context_id" : "101" ,
11 "media_type" : "application/oracle.com.cloud.common.VM+json" ,
12 "service_family_type" : "iaas" ,
13 "created" : "2016-02-24T13:12:31"
```

Figure 3.6: Example of a response to the HTTP POST request to create a VM that is shown in Figure 3.5.

VM and the second number, 1000, specified the speed of each CPU in MHz. In line 3, the value for memory is 4000 MB and the list of parameters in lines 4–6 is used by the Oracle IaaS cloud manager to configure the VM including naming its clusters and specifying root passwords. The latter may be a security breach if the HTTP request is send as unencrypted cleartext. Using IaaS cloud interfaces, it is possible to create a program that scales out the application deployment based on the factors that are beyond the control of the cloud provider.

An example of the HTTP response message is shown in Figure 3.6. In lines 1 and 2 the path to the created VM is given and its allocated name, respectively. The key `uri` stands for *Uniform Resource Identifier*, a sequence of bytes that uniquely identifies a resource on a network. The reader is already familiar with *Universal Resource Locator (URL)* notation that in addition to a URI for a resource on the *World-Wide Web (WWW)*, specifies how this resource is accessed (e.g., via HTTP when the URL starts with http:// or its *Secure Socket Layer (SSL)* version https://) and where it is located on the network. For example, the URL `http://www.cs.uic.edu/Lab/Service.html` specifies that the access mechanism is the HTTP protocol that is used to access the server `cs.uic.edu` located on the WWW and the resource `/Lab/Service.html` is accessed on this server. Finally, a *Universal Resource Name (URN)* specifies specific namespaces for accessing a resource (e.g., to access a map data object, a URN may look like the following: urn:gmap:lattitude:41.8752499:longitude:-87.6613012). Both URL and URN are subset of the URI.

In lines 3–9 the state of the created resource is given along with additional information, such as the date of the request and the creation data. The context of the created VM is given in line 10, which is a number that the client will use to send requests to a specific VM. The protocol that the client should use to communicate with the cloud manager about the created VM is given in line 11, the service type is IaaS, which is specified in line 12, and the time of creation of the VM is given in line 13. Using this information, the client can submit further requests to the cloud manager to control the lifecycle of the VM.

The IaaS model offers great flexibility in creating, controlling, and maintaining software applications in the cloud. Of course, given more level of control, it is the responsibility of the cloud customers to ensure that they develop and test software applications in a way that will allow them to extract significant benefits for a large number of available resources. Since cloud providers charge for the usage of hardware resources, inefficient implementation of IaaS or errors in allocating VMs may lead to significant charges while the performance of the application may be very poor. In addition, customers are responsible for securing their data, which is in itself a daunting task. Cloud providers, on the other hand, must ensure that they provide fast network and appropriate hardware on demand. For example, one large financial high-speed trading company purchased an extra-service at a high premium from a cloud provider to ensure reliable and fast network communication. The cloud provider enabled two separate fail-over networks for this customer, however, the network service failed during an important trading session. Further examination revealed that these two separate network from two separate independent providers used the same underlying physical cable without knowing about it. This cable was the weak link that led to the disruption of the service. It is needless to say that it is not enough simply to rely on the verbal assurances, where every aspect of the underlying hardware should be examined when using IaaS.

3.5.6 Function as a Service (FaaS)

FaaS is a relatively new addition to cloud service models where it realizes the idea of *serverless* cloud computing. In general, in SaaS, PaaS, and IaaS, VMs are instantiated that host software components, called *servers* that respond to requests from *clients*, which are hosted in other VMs or outside the cloud altogether. Since activation of a VM takes time and resources and many software components require non-negligible initialization time (e.g., connect to a database, load some configuration data), redundant activations should be avoid whenever possible. A key prerequisite is that the server components must either be activated to respond to requests from their clients or in lazy activation, they are started when the first request from a client arrives to the cloud. Thus, with lazy activation, if no client needs a service, then resources are saved by not activating a VM that hosts it, or once activated, the server stays on for a long time to avoid the overhead of its repeated activations and passivations.

Question 13: Explain FaaS in terms of SaaS or PaaS.

Consider the following example of collecting measurement of electric power con-

sumptions at a large scale. *Smart meters* are electronic devices that records consumption of electric energy at designated intervals and sends this information to a power provider for monitoring and billing [38]. In the city of Chicago, over four million households, companies, and organizations will eventually have smart meters installed to monitor their power consumptions. Each meter sends hundreds of bytes of power data to the cloud to process it every 30 minutes or so. It means that smart meter clients send approximately 50Gb of data every day to the cloud for processing. As the data arrives, it should be checked for correctness before submitting the data to various applications for processing. For instance, the power consumption measurement should not be a negative, a very large or a very small number. If the measurement falls within the suspicious values range, then it should be excluded from further processing and a technician should receive a notification to check the smart meter to see if it functions correctly. In some cases, the diagnostics can be automatically run remotely.

One way is to start a server in many VMs that receive the smart meter data, check its correctness, and send the data to other servers for processing. Since the data may come in short bursts followed by periods of lull, the servers may stay idle for some time. Yet, the energy provider who owns this application will pay for all VMs continuously, even though they may be idle more than 50% of their time. Starting VM takes anywhere from seconds to minutes, and once the instantiation starts, cloud providers routinely charge VM owners for at least one hour. Therefore, it does not make sense to passivate VMs, and even more, processing a unit of data from a smart meter may take only 100 milliseconds whereas activation of the VM will take much longer resulting in poor performance of the application.

To address this problem, a FaaS model is proposed rooted in functional programming, where functions are immutable first-class objects that can be passed as parameters to other functions and returned as values from them. Suppose that the incoming smart meter data is defined as a list in Scala `val l = List(1,-1,2,-1,1)`. We define a function that checks the correctness of the values as the following: `def f(v: Int)=if(v <= 0)SendDiagnosticsMsg()`. Then, checking the data is realized in the function `map` as the following: `l map(_ => f(_))`, where map takes each item, `_`, from the list, `l` and applies the function `f` to this item. No server is need to do that. FaaS is implemented as *Lambda* in Amazon Web Services (AWS) and as *Functions* in the Microsoft Azure Cloud and in the Google Computing Engine (GCE). The idea is that the cloud infrastructure can "react" to data items that arrive from smart meters and it will invoke the function `map` that applies the function `f` to the data without a heavyweight VM initialization. Cloud providers usually allow Lambdas and Functions to run in specialized lightweight VMs that take microseconds to start and customers are charged in increments of 100 milliseconds of the execution time. It is typical that cloud providers limit the execution times of functions to five minutes or to less than 10 minutes. Pricing models are somewhat complicated in general, with free first hundreds of thousands of invocations, and then a few cents per a hundred thousand requests and one millionth of cent per few Gb of data transfer per second.

Programmers create functions in languages like Java, Scala, F#, JavaScript, Python, or C# and upload them to the cloud in compressed files where these functions are registered with specific event triggers. For example, `cron` jobs are triggered at specific times and a function may be supplied as a job. Functions may be supplied as a part of a

workflow, a graph where nodes designate specific computations and edges specify the directions in which the results of the computations are sent. For example, a financial workflow at a large company maintains financial information that is received from various sources and invokes functions that may keep track of work hours or compute taxes on each purchase. These functions can be created and updated by developers who then load them up to the cloud to plug into the existing financial workflow.

Of course, invocations of tens of thousand of functions for a a dozen of milliseconds each creates a debugging nightmare, since adding logging statements to functions will add significant overhead that negates the purpose of FaaS. To address this issue, AWS introduced the notion of *step functions*[6], each of which performs a specific limited service in a workflow. The AWS documentation states: "Step Functions automatically triggers and tracks each step, and retries when there are errors, so your application executes in order and as expected. Step Functions logs the state of each step, so when things do go wrong, you can diagnose and debug problems quickly." More information can be found on AWS[7].

Finally, not only is FaaS used to create applications from functions, but it is also used within cloud infrastructures Amazon cloud services to trigger events for applications, to test whether resource configurations comply with some pre-defined rules, and to respond to certain system events, e.g., AWS CodeCommit and CloudWatch Logs. That is, with FaaS, cloud computing service providers eat their own food, so to speak, by using the functional services provided by their platforms to improve the performance and the functionality of their cloud computing platforms further.

3.6 Summary

In this chapter, we reviewed different cloud models and their operational environments. We explain the basic model of cloud computing where two fundamental properties (i.e, pay-as-you-go and resource provisioning on demand) separate cloud computing from other types of distributed computing. We described issues with utilization of resources in the cloud and showed how different cloud providers address the issues of over- and underprivisioning resources for applications. After introducing the notion of a software stack, we explained three cloud deployment models, SaaS, PaaS, IaaS, and FaaS in terms of accessing and controlling software layers. In addition, we gave examples of client programming to access services offered by the cloud software stacks.

[6] `https://aws.amazon.com/step-functions`
[7] `https://aws.amazon.com/serverless`

Chapter 4

Remote Procedure Calls

Deploying software applications in VMs in the cloud means that the users of these applications access them via the Internet to perform computations by supplying some input data to procedures that are located in these VMs and obtaining the results of these computations. In general, a procedure or a function is a basic abstraction by parameterization that is supported by a majority of programming languages. Instead of copying and pasting code fragments and assigning values to their input variables, the procedural abstraction allows programmers to specify a code block once in a named procedure, parameterize it by designating input variables, and invoke this code by writing the name of a procedure with values that correspond to the input parameters. Despite seeming simplicity of invoking a procedure, it is a complicated process that involves the operating system interrupting the execution flow of the application, locating the code for the invoked procedure, creating a special structure called a frame to store various values and instructions, executing the code of the procedure, and once it is finished, removing the frame from the memory and returning to the next instruction in the execution flow. In this section, we will analyze how to change the process of local procedure invocation to make it work in the distributed setting, so that clients can call procedures that are located in VMs in the cloud from their local client programs.

4.1 Local Procedure Calls

In 1936, Dr. Alan Turing introduced the notion of an abstract machine as a mathematical model of computation that later became known as the Turing machine. The model includes an infinite tape with a magnetic head that can move along the tape and read and write ones and zeros, i.e., bits. An implementation of the Turing machine can be viewed as an elementary computer without any procedure calls. Interestingly, Dr.Turing's Ph.D. advisor was Dr.Alonzo Church, who is known, among many of his contributions, for λ -calculus, a computational model based on function abstraction. Despite the strong influence of the notion of functions on the theory of computing, the first general-purpose computer created in 1943 and called *Electronic Numerical Integrator And Computer (ENIAC)* was a hard-wired device that was operated by setting

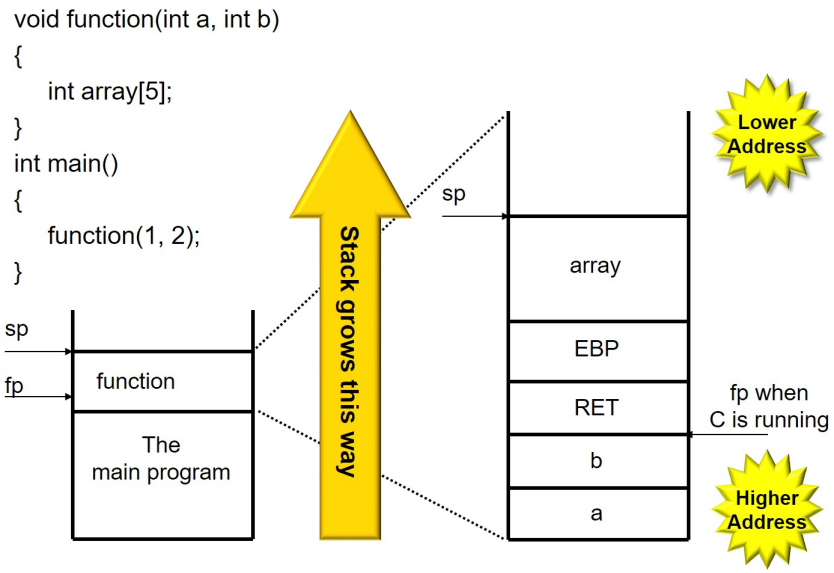

Figure 4.1: An illustration of the local procedure call.

physical dials and switches. It did not allow first programmers to create procedures and call them from programs.

Economically, it did not make sense to build computers where instructions could be changed only by physical rewiring. In 1944, ENIAC inventors, John Mauchly and J. Presper Eckert proposed the design for the next generation computer called *Electronic Discrete Variable Automatic Computer (EDVAC)* and they were joined by John von Neumann who formulated the logical design of the stored-program computer that is later became known as the von Neumann architecture. The key idea of the architecture was to separate the CPU and memory – programs would be written separately as sequences of instructions, which will be loaded and executed by the CPU and the data will be stored in the computer memory and manipulated by the instructions. Doing so represented a radical departure from hard-wired computers, since many different programs could be executed on general-purpose computers.

With the separation of the CPU and the memory, the latter is divided into five segments: the code segment that contains program instruction and the program instruction register points to the next instruction that is to be executed by the CPU, the data segments where initialized and static values of the program variables are stored, the *Block Started by Symbol (BSS)* segment that stores uninitialized program variables, and most importantly, the heap and the stack segments. The heap stores program variables that can grow and shrink in size, whereas the stack is the temporary storage for the program execution context during a procedure call. The stack is controlled by a special operating system subroutine called the *calling sequence* or the *activation record*,

An illustration of the local procedure call is shown in Figure 4.1. We base our description on the chapter of the Intel documentation that described procedure calls,

interrupts, and exceptions. Like many other computer architectures, Intel uses the procedure stack to reflect the *Last In First Out (LIFO)* order of procedure calls, where the last called procedure will finish and return the control to the calling procedure. In the upper left corner, the source pseudocode is shown where the procedure `function` is called from the procedure `main`. Underneath it, in the lower left corner the stack is shown with the calling procedure `main` on the bottom and the called procedure `function` on top of the stack. In Intel 64-bit architecture, the stack can be located anywhere in the program memory space and it can be as large as a single segment, which is 4GB.

Recall that a stack is a LIFO data structure with two operation, `push` and `pop`. Once a call is made to the procedure `main`, it is pushed on the stack, the processor decrements the *stack pointer (SP)*, since the stack grows from the higher to the lower addresses. When the procedure `function` is called, the operating system allocates a new frame on the stack for this procedure on top of the frame for the procedure `main`. Only the top frame of the stack, i.e., the procedure `function` in our example, is active at any given time. When the last instruction of the top frame procedure is executed, its frame is popped and the processor returns to the execution of the next instruction in the previous stack frame procedure that becomes the top stack frame. Thus, the configuration of the stack is dictated by the control flow of the program instructions that call procedures.

Question 1: is it possible to design an algorithm to predict the stack configuration at any given execution time for an arbitrary program?

To ensure that the processor returns to the last instruction before calling a procedure, and to provide the context for a called procedure, a frame should keep certain information in addition to the general bookkeeping information for the stack. To link the calling and the called procedures, the processor keeps track of SP and return instruction pointer. The processor stored the SP value in its base pointer register, so that the processor can reference data and instructions using the offset from the SP value in the current memory segment, making address computations very fast. To obtain return instruction pointer, the call instruction to a procedure pushes the current instruction address in the Instruction Pointer register onto the current stack frame. This is the return instruction address to which the control should be passed after the called procedure is executed. If the return instruction pointer is damaged or lost, the processor will not be able to locate the next instruction that should be executed after the called procedure is finished. It is interesting, though, that the Intel documentation states the following fact: "the return instruction pointer can be manipulated in software to point to any address in the current code segment (near return) or another code segment (far return). Performing such an operation, however, should be undertaken very cautiously, using only well defined code entry points."

Question 2: what is the relationship between the stack overflow problem and storing the return instruction pointer?

The expansion of the stack frame for the procedure `function` is shown on the right side of Figure 4.1. The layout of the stack frame differs among different computer architectures, so we concentrate on main principles in our description. Parameters to procedures can be passed through registers, as a pointer to the argument list, which is placed in a data segment of the memory, or by placing these parameters on the stack. The called procedure can return results exactly the same ways. In our example, the arguments (i.e., the variables `a` and `b`) are placed on the stack followed by the returned address, the BP register value, and the local variable (i.e., the array of five integers, `array`). The calling context, i.e., the state of the calling procedure and the content of values in hardware stores (i.e., registers and some memory locations) should be saved and restored after the called procedure finishes its execution. To complicate things further, procedures can be located at different privilege levels, i.e., in the Intel architecture, level 3 for applications, levels 2 and 1 for the operating system services, and level 0 for the kernel. Handling those procedure calls involves checking the access level by the processor and performing somewhat expensive operations for creating a new stack and copying values from the previous stack to the new one, in addition to complicated manipulations of values across multiple registers and memory locations.

Moreover, interrupts and exceptions can happen when executing a procedure, where an interrupt is an *asynchronous* event that is triggered by an *Input/Output (I/O)* device and an exception is a *synchronous* event that is generated by the processor when some conditions occur during the execution of an instruction (e.g., division by zero). By saying that an event (e.g., creation of a data structure) is asynchronous, we mean that this event is generated at a different rate or not together with the instructions that the processor is executing at some moment and the process that generated this event does not block. However, a synchronous event occurs at some predefined time with some other event (e.g., a phone call is answered synchronously after the phone rings at least once). When an interrupt or an exception occurs, the processor halts the execution of the current procedure and invokes a special interrupt/exception handler procedure that is located at a predefined address in the computer memory. Depending on the type of the event, the processor will save the content of hardware registers and some memory fragments, obtains a privilege level for the handler, and saves the return address of the current procedure before executing the handler. Once finished, the processor returns to the previously executing procedure on the stack. Those who are interested in more detailed information, can find it in the Intel 64 Architectures Software Developer's Manual [1] or in the documents for the corresponding architectures.

So far, we have used the terms synchronous, asynchronous and (non)blocking to describe function calls. Let us refine the distinction from the perspective of executing I/O operations, e.g., reading from or writing into a file or a network connection. A synchronous I/O function call to retrieve data blocks until some amount of the data is retrieved (or whatever data is waiting in some buffer for a nonblocking call) or an exception is returned informing the caller that the data is not available. This function call is also *blocking* since the client waits some time for the completion of the function call before proceeding to the next operation. Conversely, an asynchronous call is a non-blocking function call returns immediately. For example, a reference to the called

[1] `http://www.intel.com/Assets/ja_JP/PDF/manual/253665.pdf`

function can be put in an internal queue and some component will eventually retrieve the information about the called function and invoke it. Regardless of the subsequent actions, the client proceeds to execute next operations and commands without waiting for any results from the called function. This function call is asynchronous, since the execution of this function does not affect the execution of the immediate subsequent operations by the client. Once the function call is executed, its results will be sent to the caller via some callback function that the client provides when making the asynchronous call.

A confusion often arises because a synchronous function call is assumed to be blocking, however, it is not the case. A synchronous I/O function call to read data from some file may return with a subset of the available data, hence it will not block to obtain the entire set of the available data. As a result, the client must check the status of the I/O handle or the return value of the I/O function call and if more data is still available, the client will keep calling this function in a loop. Thus, such function call is both synchronous, since the client waits to get some result back and nonblocking, since the caller is not blocked for the entire duration to get all results back. By definition, asynchronous function calls are always nonblocking,

4.2 Calling Remote Procedures

We call a procedure remote if it is located in a disjoint address space from a calling procedure. That is, the calling procedure contains instruction `call` < *address*> whose semantics is to instruct the OS to invoke activation record for the called procedure whose first instruction is located at the address *address*. Once a new frame for the called procedure is created, the CPU executes its instructions starting with the one located at the address *address*. Of course, if all procedures are located in the same address space, then passing the procedure address to the CPU is a simple technical matter and the CPU will use the address to invoke the instructions at the target procedure.

This situation is aggravated if the called procedure is remote, e.g., it is located on a different computer. The address *address* is not valid any more, since it belongs to the memory space of the calling procedure that does not share the memory space of the remote procedure, i.e., their sets of memory locations are disjoint. Even if both calling and remote procedures are located in the same physical memory on a single computer, the calling procedure may run in one process and have no access to the memory locations owned by a different process that hosts the called (remote) procedure. Modern CPU architectures and OSes implement memory protection to control which process has access to what memory locations. For example, the virtual memory management includes special OS routine with hardware support that map virtual memory pages to physical addresses and determine which processes own these pages. If one process attempts to call a procedure directly by its address that is owned by some other process, an OS exception will be thrown.

Question 3: Why is protected memory needed? Wouldn't it be simpler if all processes could read and write all memory locations asynchronously?

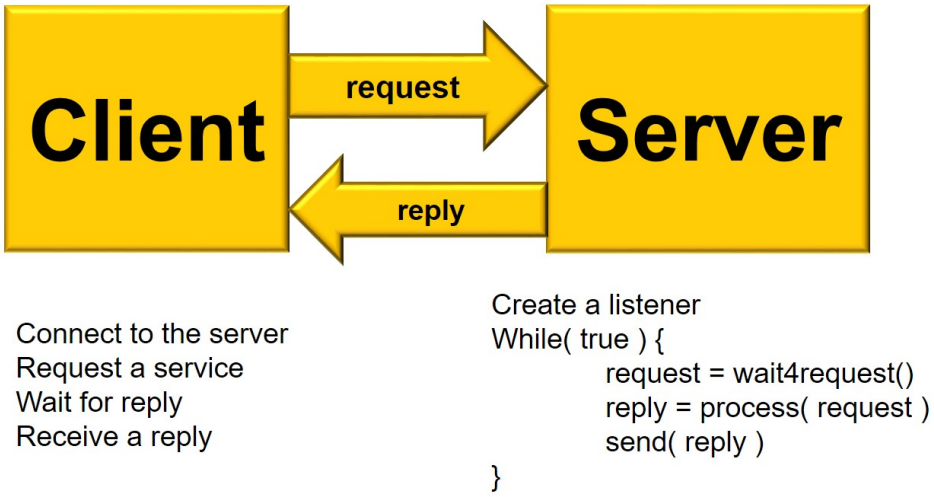

Figure 4.2: A model of the client/server interactions when calling remote procedures.

To overcome this problem, messages are passed between the process where the call is made to a remote procedure and the process that hosts the called remote procedure. A message is a sequence of bits in a predefined format that contains information about the caller and the callee as well as the values of the input parameters and the results of the computation. Consider a client/server model of remote procedure calls that is shown in Figure 4.2. The client is the process in which a remote procedure call is made and the server is the process where the actual procedure is hosted. Communications between the client and the server is accomplished using messages, which are sequences of bits formed according to some protocol (e.g., TCP/IP). The message from the client to the server is called a *request*, and the message that the server sends to the client is called a *reply*. These messages are shown with block arrows in Figure 4.2 and the explanations of the functions of the client and the server are given below their corresponding boxes.

The server creates a listener to receive requests from its clients and it waits for these requests in the WHILE processing loop. The client process connects to the server and it runs independently from the server process until a remote procedure call is made. At this point, the call is translated into a request for a service that the client sends to the server. The service is to invoke a procedure with the specified values of the input parameters. Once this request is sent, the client waits for a reply from the server that contains the result of the computation or the status of the finished execution if the function does not return any values. This is the simplest semantics of the client/server model of the remote procedure call.

Question 4: How to handle exceptions that may be thrown during the execution of the remote procedure at the server side?

We can view exceptions as objects that are created by the execution runtime in

response to instructions that result in some incorrect state of the application or the runtime environment itself (e.g., a disk error that causes the filesystem to issue an exception). Once an exception occurs, the control is passed to exception handlers, which are procedures that the runtime invokes automatically in response to certain exceptions. Suppose that an exception handler is defined in the client and the exception is thrown in the server. It is unreasonable to expect programmers to write complicated code that detects the creation of exception objects in the server program and transfers the information about them to the client program. Clearly, this process should be automated and programmers should reason about the program as a whole.

Question 5: Since this RPC model is based on message passing, one immediate question is how it is different from using some *interprocess communication (IPC)* mechanism to make a remote procedure call?

In general, significant manual effort is required to create programs that use IPC to send data between components that are run in separate process spaces. Not only must programmers define the message format, write the code that sends messages and receives them and locates and executes the remote procedures, but also they must handle various error conditions, e.g., resending messages that are lost due to network outage. In addition, an IPC mechanism on a Linux platform may have different interfaces and some differences in its semantics from the Windows, and it will result in multiple implementations of the IPC-based RPC for different platforms. As an exercise, the reader can implement RPC using sockets or named pipes for a Linux and the Windows platforms to understand the degree of manual effort, which we call low-level implementation. Opposite to low-level is a high-level coding where the programmers reason in terms of procedure calls rather than establishing physical connections between the client and the server and sending messages using these connections.

Question 6: Is RPC language-dependent?

A software tool is language dependent if its use differs from a language to a language. Constructing a message according to some format is language-dependent, since the code that (un)marshalls bits in a message uses grammatical constructs of the language in which the code is written. Written in one language, the code fragment cannot be easily integrated into a program written in some other language without changing the code. Opposite to it, a language-independent implementation of the RPC does not require the programmer to write RPC-specific code in some language. We will discuss how to achieve language independence for RPC in more detail in Section 4.5.

Question 7: What information should be put in clients and servers, and what is the exact content of the messages that they exchange?

Clearly, when the client requests a service, it should specify the address of the server that responds to requests from its clients. The server address could be a unique

sequence of bytes like an IP address or an internal network address within an enterprise network. A version of the server may be included in the address. If the server listens on a port, then it should be included in the address. In summary, all information that specifies the network address of an RPC server process is called an *endpoint*. Many endpoints can be abstracted by *endpoint mapping*, a process that manages a pool of endpoints and assigns them to specific RPC services, sometimes, dynamically on-demand. Doing so helps servers utilize its resources more effectively by maintaining a smaller pool of endpoints and mapping it to specific remote procedures.

One solution to the problem of invoking a remote procedure as if it is a local one is to reconcile disjoint address spaces. That is, a programmer is given a view of a contiguous single memory space, which is a union of disjoint address spaces, however, in reality, these spaces remain disjoint. This single memory space is called *distributed shared memory (DSM)* that enables programmers to read and write data in the distributed global address space [109, 126]. An implementation of this concept involves instances of a memory manager in each disjoint address space, and these instances ensure the coherence of memory views for all users.

One of the main benefits of the DSM is that programmers do not need to construct, pass, and receive messages among disjoint memory spaces, since they can invoke methods simply by specifying their names, which are bound to specific addresses in designated address memory spaces automatically using the memory manager instances. Since there is no physical hardware bus that connect disjoint memory spaces, there is no bottleneck that may result from transferring multiple data along this bus. Moreover, computers that run different OSes and that are located in various parts of the world can be joined in the DSM as long as they run memory managers that can offer this useful abstraction.

Unfortunately, DSM has many disadvantages, and in order to understand them one has to review some algorithms that achieve the DSM abstraction. Suppose that there are multiple computing nodes, $N_1, \ldots, N_s$ and each node has its own memory address space. On the other hand, we have a user who is presented a shared virtual memory space where these specific computing nodes are abstracted away. To map the global address space from the shared virtual memory to specific computing nodes and vice versa, each of them runs a *shared memory manager (SMM)* that perform the mapping. One algorithm uses the central server that receives memory reads and writes from users and maps these requests to specific computing nodes. Another is a replication algorithm that makes copies of data objects to multiple computing nodes and SMMs keep track of their locations. There are many DSM algorithms like that, and they have one common issue – memory coherence, where the value returned by a read request issued by a user to a memory location in the virtual address space should be the same value that is the last written value to this memory location before the read request.

Unfortunately, achieving full memory consistency in a DSM is a very difficult problem. Consider a situation when a user writes data to some virtual memory location that is mapped to a concrete memory cell at the computing node N_k. Since the virtual memory is an abstraction and not a physical resource, it cannot hold data. The write request is forwarded to the concrete memory cell at the computing node N_k and it does not happen instantly. The time it takes between the moment that the DSM receives a request and the moment it accesses the memory cell on a specific computing node is called

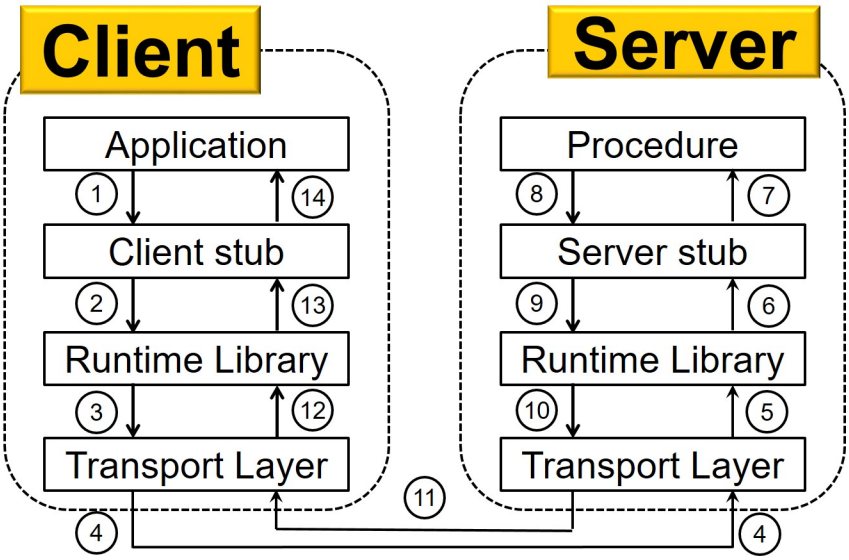

Figure 4.3: The RPC process. Circles with numbers in them designate the order of the steps of the process with arrows showing the direction of the requests.

the request *latency*. Suppose that a different user issued a read request to the same DSM memory location after the previous write request and this read request is mapped to some other computing node, N_p where $k \neq p$. Since the write request latency is non-zero, the memory of the node N_p will not match the memory of the node N_k for some time. It means that the memory will not be coherent and the user's read will not return the value that was written to the same memory location prior to the read. DSM algorithms attempt various trade-offs between the level of memory consistency and the performance. As we shall see in later chapters of this book, the issues of latency and data consistency are key to engineering cloud applications.

4.3 The RPC Process

A model of the RPC process is shown in Figure 4.3. The dashed rectangles designate the client and the server and the solid rectangles specify their components. The communications are designated with directed arrows and the order of the communications in this process are depicted with numbers in circles next to the corresponding arrows. This model is common for all implementations of the RPC process [25].

The client application executes until the location is reached when the procedure is invoked using its name with arguments to it passed as if the call is local. This call is made to the client stub ①, where the implementation of the procedure contains the code that initiates a network connection with the server, marshals the values of the arguments using a predefined format, and ② passes the marshalled message as parameters

to a call from the RPC runtime library, which contains various functions for data conversion, exception handling, and remote connectivity using various IPC mechanisms. In some cases, the RPC runtime library can batch multiple remote procedure calls into batched futures to sent multiple procedure calls in one batch to reduce the communication overhead between clients and servers [27]. The transport layer ③ receives the message and transmits it to its counterpart on the server side ④ that ensures that the message is received and assembled correctly if it was broken into packets on the client side. Once the transport layer on the server side receives the message, ⑤ it passes it as the parameter to a call from the runtime library, where the message is decoded, the destination remote procedure is identified, and ⑥ the server stub for this procedure is invoked that contains the implementation of the invocation of the actual procedure and how to pass arguments to it. Thus, ⑦ the server stub invokes the actual procedure that contains the code that a programmer wrote to implement some functionality. Doing so completes the client to server call of the RPC process.

Once the procedure finishes, the results of this execution should be passed back to the client. Recall that the client can wait until the remote procedure finishes its execution to proceed with its own (i.e., the synchronous RPC) or it can proceed with its execution and the results of the execution of the remote procedure will be delivered independently to the client's address space and written into the designated memory locations (i.e., asynchronous RPC). In both cases, the RPC process is the same. Once the procedure terminates, ⑧ its return value is passed to the server stub that marshals the values ⑨ to pass to the RPC runtime library. Even if the remote procedure does not return any results, the return is considered an empty value of the type `void` and it is passed to the client. The runtime library ⑩ passes the return to the transport layer that in turn ⑪ passes it to the client side ⑫ → ⑬ →⑭ back to the location in the source code where the remote procedure call was invoked and it is treated as if it came from a local call. This completes the RPC process.

4.4 Interface Definition Language (IDL)

The RPC process that is shown in Figure 4.3 raises many questions. How are the client and the server stubs implemented? How is the server located? What happens if the client invokes a procedure with different arguments that are expected on the server side? What happens if the client invokes a procedure that does not exist at all on the server side? Or when the type of the return values is different from the type of the local variable into which this return value is stored? How to evolve the client and the server by adding new procedures and modifying existing ones without introducing errors? More importantly, how to create an RPC application where clients are written in different languages from the one used to write the server? These and other similar questions point out that there must be some enforceable contract between the client and the server to ensure a certain level of the correctness of the RPC-based applications.

Question 8: Argue pros and cons of adding definitions to the methods of interfaces in IDL.

```
1 [uuid(906B0CE0-C70B-1067-B51C-00DD010662DA), version(1.0)]
2 interface Authenticate {
3         int AuthenticateUser([in, string] char *Name);
4         void Logout(int);}
```

Figure 4.4: IDL code example of the authentication interface and its remote procedures.

A conceptual outline of the solution that addresses these question is to abstract the specification of the interactions between the client and the server in the client/server RPC model via the concept of the *interface*. Instead of revealing all details about the implementation of the remote procedure, the client has information only about the interfaces that the server exposes, and these interfaces include the names of the remote procedures, their signatures that include types and the order of their arguments and the types of the return values. Using interface definitions, developers can create clients independently of the remote procedures and in parallel to their creation.

This concept was realized in the Interface Definition Language (IDL)[2], whose main idea is to create a language- and platform-independent way to creating RPC-based software while enabling programmers to use type-checking facilities of the target language compiler to catch errors at compile time. A key idea behind the IDL is to introduce a standard for creating interface specifications and then to use created specification to generate the implementation of the stubs for desired platform and a target language, thus burying the complexity of writing custom code in the tool that generates this code and allowing programmers to concentrate only on the main logic of the application they develop and avoid reasoning about the low-level code that realizes the RPC protocol.

Consider the RPC development process that is shown in Figure 4.5 whereas an example of IDL is shown in Figure 4.4. The annotation in the square brackets specifies the unique identifier (i.e., `uuid` stands for universally unique identifier) of the interface and its version. The identifier can be viewed as a reference to the RPC server. The uniqueness of the identifier is guaranteed by the algorithm that it generates by using random seeds, information from the computer hardware on which this algorithm runs, time, and other values that the algorithm combines into an identifier. Using the same algorithm, it is highly unlikely that the same identifier will be generated unless the algorithm runs nonstop for a couple of billion years. Assuming that the combination of the `uuid` and the version uniquely defines an interface, locating it becomes a technical matter for an endpoint mapper.

The rest of the code in the IDL example define the signatures of two procedures: `AuthenticateUser` and `Logout`. An important point is that there is no implementation of a procedure is allowed in IDL, only its definition. In fact, it is up to the application programmer to define and implement algorithms that realize the semantics of the defined methods. The IDL file marks the beginning of the RPC development process, where it is ① processed by the IDL compiler as shown in Figure 4.5, and it ② outputs the header file, `.h`. We can add two more inputs to the IDL compiler: the target language and the destination platform, which we tacitly assumed to be C and Unix. The output header file can be viewed as a conversion of the IDL interface definition into the target language and it is used in the subsequent steps of the development

[2] `http://www.omg.org/gettingstarted/omg_idl.htm`

The IDL compiler also ③ generates the client and the server stubs that are shown in Figure 4.5 with the `_c.c` and `_s.c` extensions of the IDL file name. Recall that stubs contain calls to the underlying RPC runtime library as well as the implementation of argument (un)marshalling and exception handling, among other things. Here the benefit of the IDL concept becomes clearer - it allows programmers to define interfaces in a platform- and language-independent way and then the IDL compiler will generate a low-level implementation of the RPC support, so that the programmers do not need to worry about implementing this support manually. That is, the IDL abstraction allows programmers to concentrate on high-level definitions of interfaces between the client and the server and invoke the remote procedures the same way as if they are local.

At this point, programmers created `ClientImp.c` and `ServerImp.c` files that, as whose names suggest contain the implementation of the client and the server programs. Some programmers may independenly implement the body of the remote procedures in the server modules while other programmers implement the client programs with invocations of these remote procedures. The generated header file is ④ included in the client and the server programs. Doing so enforces the type checking procedure that verifies that remote procedure calls adhere to the same signatures defined in the IDL specification. With this type checking, the language compiler can detect errors in the incorrect usage of the remote procedures statically.

Once the client and the server programs are written, the target language compiler and linker ⑤ generate the executable code that is ⑥ linked with the stub implementations and the runtime RPC libraries that are provided by the given platform. The client and server programs ⑦ are generated and can be deployed independenly thus completing the RPC development process.

4.5 Pervasiveness of RPC

RPC is one of the most successful concepts that survived the test of time: it has been around for over 30 years, it was heavily researched, implemented, and used in many software projects. RPC is an integral part of the operating system bundles, and many OS services depend on the presence of the RPC runtime libraries. Modern implementations of big data processing framework like Map/Reduce use the RPC under their hoods. Moreover, new RPC implementations mushroom: Facebook uses Apache Thrift and Wangle in its cloud-based applications, Google implemented and made publically available gRPC, and Twitter did the same with its RPC implementation called Finagle. We will review these tools and approaches in the next chapters.

> **Question 9**: Please give examples of applications where the explicit use of low-level interprocess communication mechanisms like shared memory is preferable to the use of RPC.

The success and pervasiveness of RPC can be viewed as a combination of three factors: simple semantics, the efficiency of the approach, and the generality of the concept. Functions are basic abstractions, where a fragment of code is parameterized

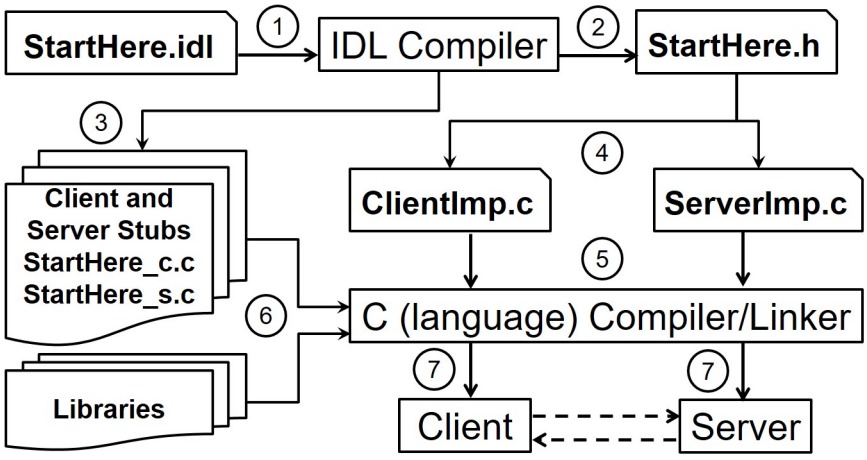

Figure 4.5: The RPC development process with IDL. Circles with numbers in them designate the order of the steps of the process with arrows showing the direction of the requests. Dashed arrows show the interactions between the client and the server.

and modularized in scope that is assigned a unique name, A function call is essentially an invocation of the code fragment that the function name designates with parameter values replacing the formal parameter names in the code (i.e., we do not distinguish here passing by values vs by reference vs by name at this point). Once the function finishes its execution, it is replaced by the return value that it computes. This simple semantics is preserved in RPC making it easy to reason about the source code.

Next, over many years, RPC implementations got very efficient. In a way, one can think of an IDL compiler choosing right code transformation techniques and algorithms to generate the client and the server stubs that carry the brunt of low-level communications that realize RPC calls. There are many ways to optimize the RPC: using different low-level IPC primitives to suit the locality of the client and the server (e.g., shared memory can be used to pass messages if both the client and the server are located on the same computer), batching procedure calls, or lazily passing parameter values by waiting until the remote procedure actually needs these values during its execution. Moreover, in the cloud datacenters with hundreds of thousands of commodity computers using highly efficient RPC is a major demand. For example, a joint work between Microsoft and Cornell on RPC chain introduces a primitive operation that chains multiple RPC invocations in a way that the results of computation are transferred automatically between RPC servers without the need for clients to make explicit remote procedure calls [145]. The implementation uses the concept of chaining functions where the results of computing of the inner chained function is used as an input parameter to the next outer function that is chained to the inner function. Using RPC chain enables programmers to create highly efficient workflows within cloud datacenters. In another work, company called CoreOS, Inc. released a product called *etcd*, a

distributed key value data store that works with clusters of computers [3]. Its documentation states: “The base server interface uses gRPC instead of JSON for increased efficiency.” That is, one of many RPC implementations such as Google RPC (i.e., gRPC) is used as core components of commercial and open-source cloud solutions nowadays.

4.6 Summary

In this chapter, we introduce a very important concept of remote procedure calls that is cornerstone of creating distributed objects in cloud computing specifically and in distributed computing in general. We start off by explaining how local procedure calls work and show that the model of the local procedure call does not work in the distributed environment. Next, we explain the basic client/server model of RPC and introduce requirements to make it powerful and easy to use. We show how RPC process works and we ask questions how to make it effective and fitting into the general software development process. We described issues with using low-level interprocesss communication mechanisms to implement RPC and we showed how adhering to the basic function invocation without using low-level calls makes source code simpler and easier to maintain and evolve. Then, we introduce the notion of Interface Definition Language (IDL), explain it using a simple example, and then describe the RPC development process with IDL. We conclude by summarizing main points what made RPC a pervasive and successful concept and implementation.

[3] `https://coreos.com/etcd`

Chapter 5

Cloud Computing With the Map/Reduce Model

The author of this book worked at a large furniture factory in the middle of 1980s in the Soviet Union where he wrote software in PL/1 that computed reports for executives about raw timber delivery and furniture production at the factory. The data were collected manually during the day from various devices and people who operated them as well as warehouse agents and logistics personnel. The amount of daily data ranged anywhere approximately between 50Kb to 200Kb, very small by today's standards, and these data were often manually reviewed, mistakes were identified and the data was adjusted accordingly, and the batch report algorithms ran overnight to produce the report summary by 9AM the next morning for executives to review these reports. The author's first job in the USA in the beginning of 1990s was to write software to collect data from infrared sensors installed on various items in a plant to track these items as they were moved around the plant. The amount of data was larger, close to 5Mb a day from a large plant and processing was done in real-time. Yet the amount of data was still very small compared to the avalanche of data that came after the Internet took off in the second half of 1990s.

Between 1990s and 2000s a few economic and technical factors changed the nature of computing. The Internet enabled production and distribution of large amounts of data. Different types of sensors, cameras, and *radio-frequency identifiers (RFID)* produced gigabytes of data first weekly, they daily, then hourly. Average non-computer-science type users created content on the Internet as web pages and social network nodes. News media started posting articles online. The data sizes were measured first in gigabytes, then terabytes and then petabytes and exabytes. This dramatic increase in the available data came together with advances in computer architecture that decreased the cost of storages to cents per gigabyte and the cost of a commodity computer to few hundred dollars from thousands in the beginning of 1990s. A new term was coined, *big data* that collectively describes very large and complex datasets that are difficult to process using a few servers even with many multicore CPUs[1].

[1] `https://en.wikipedia.org/wiki/Big_data`

5.1 Computing Tasks For Big Data Problems

It is not just the size of the data that matters and how cheap it is to store it and move it around, but also the changing nature of the computing tasks that are performed on big data. Consider the PageRank algorithm that is at the core of the Google search engine [88]. Without getting into details of this algorithm, it operates on the Web connectivity graph where nodes are web pages and directed edges specify relations between these pages when one page references some other page using, for example, the HTML hyperlink operator `href`. A goal of the algorithm is to compute the probabilities that show how likely a user can reach a web page after s/he starts browsing from some other web page. The Web connectivity graph is represented as a square matrix with $n \times n$ dimensions, where n is the number of web pages. Each row corresponds to a web pages and so does each column. Each cell in the matrix designates the connectivity between the corresponding web pages that are represented by the row and the column that intersect at the cell. Naturally, the diagonal cells are set to 1, since each page is connected to itself. The value in a cell can be between zero and one, designating the probability of reaching a web page designated by the column from some other page designated by the row of the matrix.

The other matrix is the matrix of the importance values of the pages that has the dimension $1 \times n$, and it is the actual page rank vector. Each column in this matrix correspond to a web page. Multiplying these two matrices results in the updated matrix of the web page importance. The algorithm is iterative, meaning that the connectivity matrix is multiplied by the result of the previous multiplications until the fixpoint vector is reached according to some termination criterion.

It is estimated that there are over 4.6 Billion web pages and assuming that each matrix cell holds a four byte floating point value of the probability, the storage required for this matrix is over 80 Exabytes or 80 Million Terabytes. More importantly, the algorithms performs many mulitplicative operations in iterations over the matrix, which may take a long time for a matrix of this size. Considering that the big-O complexity of the matrix multiplication is $O(n^{2.(37)})$ for the fastest matrix multiplication Coppersmith-Winograd algorithm and assuming that it takes only one microsecond to compute the value for each pair of the elements of the multiplying matrices, it may take over 650 years to compute the resulting product of the matrices. This example illustrates that viewing the computation of PageRank through the lenses of simple matrix multiplication algorithm is not sufficient in this context.

A key observation of the organization of the Web is that the majority of web pages are independent of one another and many of them do not have direct links to each other. PageRank can be applied to a small subset of web pages independently from other pages to compute the probabilities of the "importance" of these web pages, and these probabilities can be merged later. Recall the concept of the diffusing computation in Section 2.1, where a computing job is split across multiple computing nodes that compute partial results and return them to parent nodes that merge results from multiple children nodes. Suppose that we have 50,000 computing nodes and we can divide the matrix into submatrices of the dimensions $92,000 \times 92,000$ with the required storage of a little over 30Gb. A matrix like this can be easily fully loaded into the RAM of an inexpensive commodity computer, and it can multiply two matrices like that in less

than nine hours. This example conveys the essence of the idea of splitting a technically impossible to accomplish computing task into manageable subtasks and executing them in parallel thus dramatically decreasing the overall computing time.

Question 1: How dependencies among arbitrary data items in a set may complicate the division of a task that uses this data into subtasks?

Consider a different simple and very common task of searching text files for some words. More generally, this task in known as *grepping*, named after a Unix command, *grep* – Global Regular Expression Print – that programmers frequently use to search text files for the occurrence of a sequence of characters that matches a specified pattern. For example, the following command "`grep '\<f..k\>'*.txt`" executed without the external double quotes prints a list of all four-character words found in your files with the extension `txt` starting with "f" and ending in "k" like `fork`, `fink`, and `fisk`. For an average Unix user, this operation will terminate in less than a minute searching through a few hundreds of text files, depending on the complexity of the pattern and the number of matches and the performance of hardware. However, executing an operation like this on $4.6 Billion web pages may take thousands of years to complete on a single computer.

Nowadays, many machine learning algorithms operate on big data to learn patterns that can be used for automating various tasks. These algorithms are very computationally expensive, they take significant amounts of RAM and many iterations to complete. For example, a movie recommendation engine at Netflix analyzes rankings of more than 10,000 movies and TV shows provided by close to 100 Million customers and computes recommendations to these customers using its proprietary machine learning algorithms. As with the previous examples, the problem is to accomplish this task within reasonable amount of time and under fixed cost. Even though these and many other tasks like that are different, the common theme is that they operate on big data that contain many independent data objects. Computing results using disjoint subsets of these data objects can be done in parallel, since these data objects are independent from one another, that is, operating on one data object does not require obtaining information from some other data object. A clue to a solution lies in the idea of dividing and conquering the problem by splitting it into smaller manageable computing tasks and then merging the results of these tasks, which are computed on cheap and easily replaceable commodity computers.

Question 2: The complexity of the insertion sort is $O(N^2)$ where N is the number of elements to sort in a container. Suppose that we can parallelize the insertion sort by splitting N elements across $\frac{N}{K}$ containers with K elements in each container, sorting these containers in parallel and then merging the sorted containers. What will be the complexity of this parallelized sorting procedure?

5.2 Datacenters As Distributed Objects

Dividing a large task into tens of thousands of smaller ones requires assigning a computer to each of these smaller tasks. Large organizations purchase commodity computers in bulk thereby reducing the cost of each computer drastically. A single user may pay around $600 per computer depending on the options, but purchasing 100,000 computers in bulk decreases the cost per unit by the order of magnitude or more. However, doing so creates new problems: how to house so many computers with effective cooling systems, how to supply power uninterruptedly, how to find and fix hardware errors (this one is simple - the whole computer is replaced with a new one), and how to provide fault tolerant Internet connectivity to these computers? An answer to these questions lies in a new organizational entity called a *datacenter*.

> **Question 3**: Explain how datacenters introduce the economies of scale in computing with big data.

Essentially, a datacenter is a housing unit for a large number of commodity computers that provides an efficient cooling system, since the heat generated by 100,000 computers is significant, reliable high-speed Internet connectivity, internal network support, and 24/7 monitoring of operations that provides failover and fault-tolerance. The massive concentration of computing power in a single datacenter underlies the concept of a commodity computer as a basic good used in computing commerce that is interchangeable with other computer commodities of the same type [124]. Thus, a datacenter can be abstracted as a single computing object - its clients can view a datacenter as such by sending computing requests to it the RPC way and obtaining the results once the computation is completed. Whereas this abstraction captures the interactions between the datacenter and its users, it is important to understand key characteristics of the datacenter that separates it from the single computing object.

Elastic parallelism. Big data sets are assumed to be independent from one another and the programs that process then have no control or data dependencies that require explicit use of synchronization mechanisms (e.g., semaphors or distributed locks). In this setting, many programs can operate on different datasets in parallel, and adding new programs and datasets can be handled by assigning processing units in the datacenter on demand to run these programs to process the datasets. Whereas programmers view the datacenter as a single distributed object that runs a single task (e.g., grep or PageRank), in reality this task is split automatically into many small subtasks that are assigned to many processing units for parallel execution. Then the results are assembled and returned to the programmer, to whom the parallel processing is hidden behind the facade of a single object with well-defined interfaces in the RPC style.

Workload optimization. Whereas a user of a standalone desktop has the full control over the resources, assigning them to different tasks optimally remains a serious problem. Even in case of a single desktop, the operating system does not always

allocate resources optimally to executing processes (e.g., thread affinity problem). In case of datacenter, load balancing, resource response time, and resource locality is used to optimize the allocation of resources to different workloads.

Homogeneous platform. The virtual object abstraction presented to users hides the possible differences in hardware and software platforms that may co-exist in a datacenter. Unlike writing a program for a desktop that runs a specific operating system and that has limitations on RAM and CPUs and disk space, in datacenter all differences are abstracted away and all users are given a single view of the distributed object with well-defined interfaces. All communications with this object are performed via the RPC that enables users/clients invoke remote methods of the object's interfaces. Within the actual datacenter, all computing units can also be the same with the same software installed on them adding to the homogeneity of the computing platform in the datacenter.

Fault tolerance. When hardware fails in a desktop, its user runs the diagnostics, determines what component fails, purchases a new component of the type that is compatible with the failed component and replaces it. Desktop failures are infrequent, whereas in a datacenter with 100,000 commodity computers, failures happen every day. Given a low price per commodity computer, many datacenters simply replace one when its hardware fails. In his presentation in 2008, a principal Google engineer said: "In each cluster first year, its typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will go wonky, with half their network packets missing in action; and the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span. And there is about a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover."[2]

Since hardware failures are the everyday fact in datacenters, it is imperative that these failures be handled without requiring programmers to take specific actions in response to these failures, except some extreme cases where the majority of computers in a datacenter are affected and they cannot run programs. If a hard drive fails on a computer that contains a dataset while a process is operating on this dataset, the fault-tolerant infrastructure should detect the drive failure, notify a technician to replace the computer and move the dataset and the program to some other available computer automatically to re-run the computation. It is one example of how failures can be handled in a datacenter while presenting a fail-free mode of operation to the clients of the abstract distributed object.

[2] `http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/`

5.3 Map and Reduce Operation Primitives

Suppose that we are given a large list of N integers and we need to compute their sum. Let's assume that each addition of two integers takes only one nanosecond (i.e., 10^{-9} seconds). If N is quintillion, i.e., 10^{18}, then it will take over 31 years to complete this operation. However, we know that each pair of integers can be added independently from every other pair, i.e., the integer elements in the list are independent. As we already discussed, we can partition the list of integers into 100,000 blocks or *shards* and perform addition of each shard independently and in parallel to the others thereby reducing the time it takes to complete the whole operation to less than half an hour. Once the sum is computed for each block, the values of these sums will be added to obtain the resulting value.

Let us consider a different example that looks more complex – grepping hundreds of millions of text files to find words in them that match a specified pattern. This operation is more complex than adding integers, but we will apply the same parallelizing solution. First, we partition the entire file set into shards, each of which contains a subset of files, and we assign each shard to a separate computer in a datacenter. Then we search for the pattern in each subset of files in parallel and output a map where the key specifies a word that matches a pattern and the values contain linked lists of pointers to specific locations in files that contain these words. Thus, a result of each parallel grepping will be this map. Next, we need to merge these maps, so that we can compute the single result. If a word is contained in more than one map, the resulting map will contain an entry for this word with the values concatenated from the corresponding entries in different parallelly computed maps.

> **Question 4**: Can the proposed parallelizing solution be applied to a movie recommendation engine? Please elaborate your answer.

Even though these two examples are completely different, there is a common pattern buried in the solutions. First, the data from shards are mapped to some values that are computed from this data. As a result, multiple maps are created from shards. Second, these maps are merged or reduced to the final result and this reduction is done by eliminating multiple entries in these disjoint maps by combining them in single entries with multiple values that can be further reduced. One interesting insight is that arithmetic operations are reduction or folding operations. Summing all integers in a list reduces it to a single value. Same explanation goes for multiplying. Merging many maps whose key sets intersect in a nonempty set reduces these maps in a single one by removing redundant repetitions of the keys in separate maps. This concept is illustrated in Figure 5.1 where the flow of key-value pairs come from the left, first into the map primitives and then into the reduce primitive.

Let us give formal definitions of the `map` and `reduce` primitive operations. A *mapper* is a primitive operator that accepts a key-value pair (key, value) and produces a multiset of key-value pairs, $\{(k_1, v_1),\ldots,(k_p, v_p)\}$. A *reducer* is a primitive operator that accepts a pair (k, {value}), where a key is mapped to a multiset of some values, and it produces a map of key-set of values pair, with the same key (k,$\{v_1, \ldots, v_s\}$) and a set of

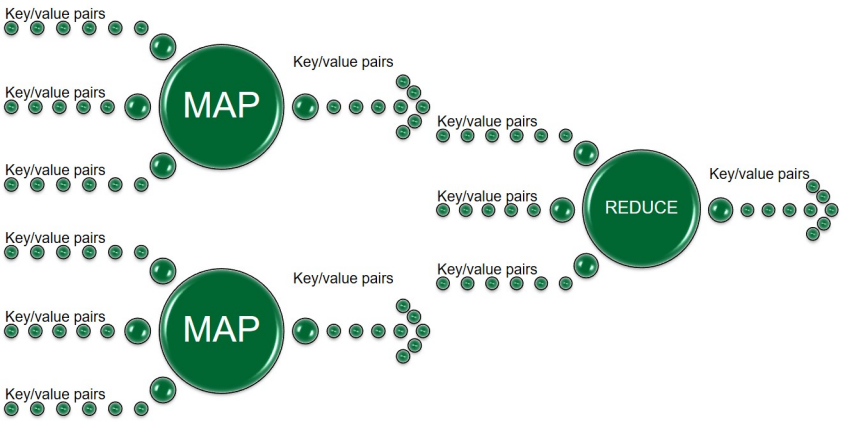

Figure 5.1: An illustration of the map/reduce model.

values that may be different from the ones in the input map. Applying these definition to the summation example above, the mapper takes pairs of integers, one as key and the other as a value and outputs a key-value pair where the key is the concatenation of these two integers with a comma in between as a string and the value is the sum of the integers. The reducer takes the the pairs outputted by the mapper and adds up their values to the total that is returned as a result of this map/reduce operation.

Question 5: Design the mapper and the reducer for the grepping problem.

A modification of the integer sum problem can make it more interesting for applying the map/reduce operation. Suppose that the goal is to determine all pairs of integers in the list that add up to the same value. In this case, the mapper takes pairs of integers, one as key and the other as a value and outputs a key-value pair where the key is the sum of two integers and the value is the concatenation of these two integers with a comma in between as a string. The reducer takes the pairs outputted by the mapper and merges them in a single map where the key designates the unique sum and the values are sets of the concatenated integers on the list add up to this sum key.

5.4 Map/Reduce Architecture and Process

Map/reduce process contains three major elements: parallelize computations using the mapper and the reducer primitives, distribute the data across multiple computing units by splitting the large input datasets into shards, and handle software and hardware failures without requiring the user intervention. The Map/Reduce architecture and the workflow is shown in Figure 5.2. Graphic elements designate distributed objects and the programmer and the arrows specify the flow of data and the sequences of operations among these objects. The barebones map/reduce structure consists of commodity computers in a datacenter, which are connected to the network, and they can communicate via the RPC. As you can see, the efficiency of RPC is important for Map/Reduce.

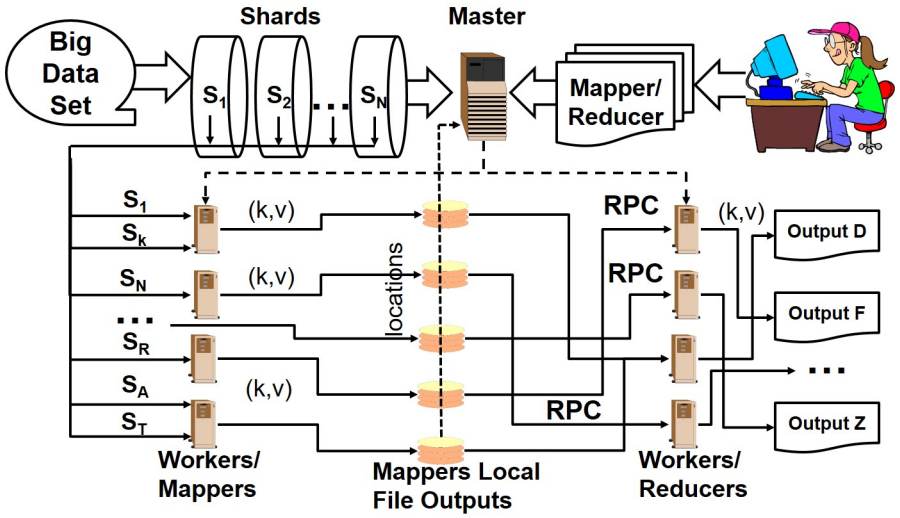

Figure 5.2: The architecture and the workflow of the map/reduce model.

Question 6: Design the mapper and the reducer for computing PageRank.

The input to the map/reduce architecture comprises the mapper and the reducer programs that are written by programmers and the input dataset on which these programs operate. The programmers' job is intellectually intensive and it involves understanding the problem, mapping it onto the mapper and reducer primitives, and implementing the solution. Once the mapper and the reducer implementations are ready, the programmers submit them to a computing node that is called the *master*, and it distributes the mapper and the reducer programs to *worker* nodes that belong to a set of designated commodity computers in the datacenter. The first step of the map/reduce process is completed.

Question 7: How many computers in a datacenter should be allocated for a map/reduce computation?

Next, a large number of computers in the datacenter is assigned to the map/reduce task. One of these computers is designated as *master*, since its job is to coordinate task distribution among the remaining computers, called *workers*. The master assigns mappers to some of the workers and the remaining workers are assigned reducers. The input dataset is split into shards, $S_1., S_N$ and the master distributes them to workers that host mappers. All communications between processes are done via the RPC. The mappers process the shards according to the algorithm that is implemented by the programmer and they produce key-value pairs that are written by the mappers in files on the local hard drives. Shards can vary in size and the original paper on map/reduce mentions the

sizes of 16Mb to 64Mb. The local disks of workers/mappers are also partitioned into regions or buckets into which the intermediate key-value pairs are written.

Once the master computer assigns shards, mappers and reducers to workers, the map/reduce computation starts. Mappers compute key-value pairs and write them periodically into the buckets on their local drives and report locations of these buckets to the master. Reducers workers do not have to wait until all mappers complete their operations – the map/reduce execution can be pipelined, i.e., the reduces can start processing the key-value pairs as mappers continue to process shards and output the pairs. In a pipelined map/reduced workflow, the output of a mapper can be fed into a reducer without waiting on the other mappers to finish. Pipelining may seem a good idea, especially considering that shards may be of different sizes and there is high variability in processing time between different mappers. This way, reducers do not stay idle waiting for all mappers to finish and the load is distributed more evenly across mappers and reducers. It would also mean more than one mapper and reducer must be used to take the advantage of pipelining the workflow.

Question 8: What can be said about the ratio of the total number of mappers and reducers to the number of the commodity computers that can be used to host them? What is the complexity of map/reduce?

So far, we observe six types of distributed objects in map/reduce. First, it is the mapper type that read in shards, which are another distributed object type. Even though shards do not contain any executable code, their default interfaces and `get` and `set`, thus allowing us to abstract both data and code into distributed object units. The master is the third type, since its job is distribute jobs among workers and coordinate the execution with the upper bound of complexity $|M| \times |R|$ where $|M|$ is the number of mappers and $|R|$ is the number of reducers, which is the forth type. Intermediate files containing key-value pairs is the fifth type and the output files is the sixth type. These six types of distributed objects are combined in the map/reduce architecture to run on commodity computers in datacenters, and this is one of the most famous examples of how big data tasks are solved by engineering distributed objects and their interactions in cloud environments by scaling out the computation.

5.5 Failures and Recovery

Since cheap commodity computers run workers and master and host all distributed objects, hardware failures directly affect computations performed by these distributed objects. Whereas implementation bugs can be analyzed using the single monolithic mode of the map-reduce implementation, where the mapper and the reducer run within a single debugger, some hardware failures are difficult to analyze in tens of thousands of commodity computers allocated for a map-reduce computation.

A key insights in failure recovery is that mappers and reducers are deterministic operations. They are also distributive (i.e., $O \odot (S_p \cup S_p) \equiv (O \odot S_p) \cup (O \odot S_p)$, where $\odot$ designates applying the operation, O that can be either a mapper or reducer to a

data object, S that can be either a shard or an intermediate key-value map file), and commutative (i.e., $(O \odot S_p) \cup (O \odot S_q) \equiv (O \odot S_q) \cup (O \odot S_p)$. That is, the results of applying map/reduce to a single dataset is the same as applying to the shards obtained from this dataset, and the order does not affect the result in which mappers and reducers are applied to the data. It means that if one operation fails, it can be retried later and in the order that was different from the originally scheduled execution.

In the Google implementation of map/reduce, the master is the single failure point that can shut down the whole computation, since it coordinates all tasks. To recover the computation more efficiently, the master writes its state into a local file at some predefined periods of time, called *checkpoints*. If the master fails, a monitor program detects that the master process does not run any more and it will restart it. The restarted master program checks to see if there is a checkpoint file, reads the last saved state, and resume the map/reduce computation from that checkpoint.

Alternatively, if a worker fails, it does not respond to a *heartbeat*, which is a control message sent periodically from the master to all workers. If the failed worker was running the mapper when it failed, the master will relocate this mapper task with its shards to some other worker. This operation is transparent, since it is hidden from users, who continue to see the computation running and eventually producing the desired result as if there were no failures. However, if the worker failed after its mapper had completed the operation, then the resulting intermediate files can be copied to a new worker. If these files are not accessible because of the disk or network failre, then the mapper should be re-executed on a new worker. Recovering from the failure that affected a reducer is even simpler, since a different reducer will pick up the input for the failed reducer and will produce the output.

5.6 Google File System

In the description of the map/reduce architecture and workflow, you can see that the filesystem is heavily used to store shards, the intermediate key-value pairs, and the outputs. Unlike desktop and laptop filesystems, human users are not expected to create and edit files manually on the commodity computers in the cloud datacenter. Apart from the OS installation and popular software distributions, users create and manage relatively small files, 10Kb to 200Kb in size, and they edit and (re)move them rarely. Hardware failures are rare on desktops and laptops, and a simple backup ensures that failures do not stop users from accessing their files. All in all, users' interactions with filesystems on their laptops and desktops does not even come close to the intensity of file processing in the map/reduce computations.

> **Question 9**: What features of the standard Unix file system are redundant for the map/reduce computations?

Big data computations heavily influenced a new set of requirements on the filesystem. Files are big and frequently changed by appending data – think of appending new key-value pairs to the input file for the reducer. Then, the pattern of file usage is

different from the one of human users – after the files are modified, they are mostly read, rather than written to. This pattern is known as *Write Once Read Many (WORM)*. With the WORM pattern, files are rarely modified, if at all. Also, concurrent usage is different – when human users share a file, one of them locks the file and the others wait until the lock is released. However, when distributed objects make the RPC to append data to a file, there is no reason to lock this file, since the RPC can be asynchronous and the data will be appended to a file in the order the requests come. Finally, the filesystem must ensure *high bandwidth* rather than *small latency*, i.e., higher bits per second transfer rates rather than smaller delays in responding to requests.

A standard example to describe the difference between the latency and the bandwidth involves cars of different speeds and capacities. A sedan with the capacity = 5 and the speed = 60 miles/hour travels faster but carries fewer passengers than a bus with the capacity = 60 and the speed = 20 miles/hour. Latencies for traveling 10 miles are the following: car = 10 min and bus = 30 min with the throughput: car = 15 people per hour (PPH), bus = 60 PPH. Thus, latency-wise, car is three times faster than bus, but throughput-wise, bus is four times faster than car.

The authors of the GFS paper say: "Our goals are to fully utilize each machines network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data." Therefore, when making a function call, some time may be spent to determine which network links to use to send the data. Doing so increases the latency. Once the network path is determined, the data is sent.

The latter point is important, since bandwidth and latency are connected. If the rate of sending bits is close or higher than the bandwidth, then the network is saturated, the data will be queued and the latency will increase. However, if the bandwidth is already high and the network is not saturated, increasing the bandwidth by adding more network capabilities will not decrease the latency. For example, satellite latency is about 20 times higher than the earth ground-bound networks at over 600 ms due to the distance between Earth and the satellite in space. In many cloud computing tasks, it turns out that the data is received and processed in large batches instead of small sizes and frequent RPC requests. This is why bandwidth takes precedence over the latency.

Question 10: Discuss pros and cons of the solution where small data blocks are batched into large ones before they are submitted to servers for processing.

Google File System (GFS) is designed to satisfy these constraints and it uses the client/server model that we discussed in Section 4.2. At the logical level, a single filesystem interface is presented as the tree with the root and branches as directory subtrees that contain other directories and files, same as in most other ordinary filesystems. On the physical level in GFS, files are divided into fixed-size units called *chunks* and they are handled by *chunkservers*. That is, a file is not mapped to a single server, its data chunks are spread across many servers. The single master server assigns each chunk immutable and *Globally Unique IDentifier (GUID)* and it maintains all filesystem metadata. Clients access files by sending requests that include chunk GUIDs to chunkservers, which run on top of Linux as user-level processes. At first, a client asks the master to provide it a chunkserver, and once the master replies with the chunkserver

address, the client communicates with this chunkserver to service its requests. Similarly to the map/reduce architecture, the master communicates with chunkservers using the hearbeat control message, and chunkservers report their states to the master. The master provides location and migration transparencies by using the metadata, which includes name resolution by mapping from file names to chunk GUIDs, access control, and the physical location of chunks, each of which is replicated on multiple computers for fault tolerance and recovery. The GFS differs from other popular filesystems in that it does not maintain metadata on the directory level. The namespace in GFS is maintained as a dictionary that maps absolute paths to metadata. The metadata is stored in RAM making it fast for the master to respond to requests.

Question 11: How do chunk sizes affect the performance of the GFS?

Keeping metadata in RAM makes it susceptible to losses due to power outages or hardware malfunction, and the master keeps and maintains persistent operation log that contains the history of metadata in a reliable storage, and this operation log is replicated on multiple independent storage units. The master flushes its memory metadata into the operation log after several modifications thus making it a checkpointing operation. The operation log is implemented as small files that contain metadata in a B-tree data structure that is a generalized binary tree where each node can contain more than two children nodes. It is fast, since insertions, deletions, and search is done in $O(log(n))$. The master is multithreaded and it creates new log files using multiple threads and writes checkpoint information in them without delaying file operations. For recovery, it uses the latest checkpoint information and old log files are purged from the system after some predefined period of time.

Question 12: How would you modify the GFS if chunks had priorities associated with the urgency of them being processed by map/reduce applications?

Recall that the main reason for inventing GFS was to make operations on big data efficient. In map/reduce, the output of mappers is appended to the files that keep intermediate results as key-value pairs, and the output of reduces is also appended to the resulting file where the data is aggregated into the final result. Making file modifications efficient is a priority for improving the overall efficiency of the map/reduce architecture. In addition, since many clients read and append data, which are replicated on many chunkservers, there is a potential for saturating the network with a flood of requests and data.

These issues are addressed in GFS by using atomic data appends and the linear network connectivity between chunkservers. The atomicity of an operation is defined as a sequence of indivisible action steps such that either all these steps occur or none does [60]. It means that appending data to a file atomically cannot be viewed as a sequence of steps where one step searches for the offset from the file head, the other positions the reference to the location where the data will be written, and the data is written in the third step. If appending data is not an atomic operation, then two concurrent data

append operations will interfere with each other when steps are executed in parallel, resetting the references and corrupting the file in the end. Explicitly synchronizing non-atomic appends requires programmers to create synchronization mechanisms that lock the file. These synchronization mechanisms are difficult to reason about and they slow down the execution, since one operation will wait until the other releases the lock.

Instead, the GFS stores data chunks in parallel without requiring programmers to use synchronization mechanisms and then automatically updates references in the metadata to designate appended data chunks to a given file. The reader can obtain in depth information about write operations from the original GFS paper as well as on the use of leases for file management and replication [60].

When it comes to replication, GFS places requirements on the connectivity between chunkservers to reduce amount of data that is exchanged between them. Chunkservers are connected in a chain forming a line where each chunkserver has two closest neighbors: one that sends data to the chunkserver and the other that receives data from it. Thus, the distance between chunkservers is measured based on the number of chunkservers through which the data from the source chunkserver will travel to reach the destination chunkserver. Enabling chunkservers to communicate only with its closest neighbors reduces the communication overhead, which could be much higher for a tree or a clique-connected network. In addition, TCP-based data transfers are pipelined, i.e., chunkservers do not wait to receive the entire data chunk, it can forward its partial packets thereby reducing the overall transfer time. Individual small gains in each data chunk transfer result in larger gains for a long map/reduce computation.

5.7 Apache Hadoop: A Case Study

The combination of the GFS and the map/reduce model led to multiple implementations, the most prominent of which is an open-source software framework called Hadoop that is maintained and distributed by the Apache Software Foundation[3]. This framework is used in many other open-source projects (e.g., Zookeeper and Spark), some of which we will review in this book. At a high level, Hadoop consists of two core components: a GFS implementation called *Hadoop Distributed File System (HDFS)* – the storage component – and an operating system called *Yet Another Resource Negotiator (YARN)* – the processing component. The first version of Hadoop contained only the HDFS and the implementation of the map/reduce model, however, in version 2, YARN was added to run on top of the HDFS to perform resource management and job scheduling. Other components are implemented and deployed on top of YARN and use its interfaces to control and manipulate data processing.

The HDFS is the main abstraction that hides the low-level details of file organization and processing on the native OS. Its key differences with the GFS is that clients can access data directly in the HDFS without asking the master for a lease on a file and the HDFS can run on many platforms, whereas the GFS is created to run on the Linux OS only. There are many other small differences like the default size of the chunk, which are implementation specific and can change in future releases.

[3] https://hadoop.apache.org

Question 13: Analyze the current documentations on the GFS and the HDFS and create a list of differences between these filesystems.

In Hadoop, two or more computers in a datacenter that are connected in a subnetwork and that run the HDFS and the YARN components are called a *Hadoop cluster*. Large clusters in big datacenters contain tens of thousand of Hadoop nodes, where a node is often viewed as a VM running daemon processes of the HDFS and YARN in the background. One or more master nodes in a Hadoop cluster coordinate the work in the cluster and worker nodes receive jobs from the master nodes and perform mapper and reducer computations. The HDFS contains three main services. The *NameNode* service that runs on the master node, maintains all metadata, and services clients' requests about the location of files. The *secondary and standby NameNode* services checkpoint metadata, and the *DataNode service* represent the GFS chunkservers that store chunks of data and update the NameNode service about the states of the local files. In the HDFS, the WORM model is used where the files cannot be overwritten, but only moved, deleted, or renamed. As the reader can see, there is much similarity in the basic architectures between the HDFS and the GFS.

Question 14: How many times are data chunks replicated in the HDFS?

Master nodes run the (subsets) of processes for`NameNode`, `ResourceManager`, and `JobHistoryServer` whereas worker nodes run the processes `DataNode` and `NodeManager`. The process `NameNode` splits the input big data file into chunks, which are stored at workers and managed by the `DataNode` processes, which in turn store, read, write, delete, and replicate data chunks. If the `NameNode` crashes, the HDFS becomes unavailable to clients, and this is why it is important to run standby `NameNode` processes. The process `JobHistoryServer` does exactly what its name suggests, it archives all job histories with their metadata for debugging and troubleshooting.

`ResourceManager` is a cluster-wide process that runs as part of YARN and it allocates resources to jobs and it controls what workers do by communicating with their `DataNode` and `NodeManager` processes. When a client submits a job to the Hadoop cluster (e.g., a map/reduce application), this job is broken into tasks (e.g., mapper and reducer tasks), which the `ResourceManager` schedules to workers in the cluster. In addition, `ResourceManager` checks the heartbeat of `NodeManager` process to determine if it is running, monitors the statuses of running jobs, manages cluster security.

YARN uses *containers* for task executions, where a container is an abstraction for resources needed to execute a given task. As a minimum, a container specifies the amount of RAM and the CPUs needed to run a particular task. It is the job of the `ResourceManager` to allocate and run containers on cluster computers.

`ApplicationMaster` is an abstract YARN process that runs in the YARN containers and it requests resources from `ResourceManager`, e.g., specific data chunks

for the job, the number and sizes of containers needed to run the job, and the locations of the containers. Concrete `ApplicationMaster` processes are framework-specific, e.g., map/reduce has `MRAppMaster` and Spark has `SparkAppMaster` processes. Once started, the `ApplicationMaster` in a container registers with the `ResourceManager` and determines and requests resources needed to run a task. The `ResourceManager` uses its scheduling component to assign containers to specific computers in the cluster. Once the `ResourceManager` grants the request to start a task, the `ApplicationMaster` will request the `NodeManager` that runs on the computer that the `ResourceManager` assigns to the task to run the container. Responding to this request, the `NodeManager` creates and starts the container (one per task) and the task runs. Once completed, the `ApplicationMaster` reports the task completion to the `ResourceManager`, deregisters itself, and exits.

Consider an example of the Java map/reduce skeleton program for the Hadoop framework that is shown in Figure 5.3. Some imports are omitted and error handling code is removed for brevity. Each program line is numbered on the left margin.

```
1  import org.apache.hadoop.*;
2  public class YourImplementationClassName {
3  public static class YourMapperName extends MapReduceBase implements
4          Mapper<LongWritable,Text,Text,IntWritable>{
5    public void map(LongWritable key, Text value,
6    OutputCollector<Text, IntWritable> output,
7    Reporter reporter) throws IOException  {
8          //here goes your implementation of the mapper
9          reporter.progress();
10         output.collect(new Text(key), new IntWritable(value));} }
11 public static class YourReducerName extends MapReduceBase
12         implements Reducer<Text, IntWritable, Text, IntWritable> {
13   public void reduce( Text key, Iterator <IntWritable> values,
14         OutputCollector<Text, IntWritable> output, Reporter reporter)
15         throws IOException {
16         //here goes your implementation of the reducer
17         output.collect(new Text(key), new IntWritable(val));} }
18   public static void main(String args[])throws Exception {
19         JobConf conf = new JobConf(YourImplementationClassName.class);
20         conf.setJobName("whatever name you choose");
21         conf.setOutputKeyClass(Text.class);
22         conf.setOutputValueClass(IntWritable.class);
23         conf.setMapperClass(YourMapperName.class);
24         conf.setReducerClass(YourReducerName.class);
25         conf.setInputFormat(TextInputFormat.class);
26         conf.setOutputFormat(TextOutputFormat.class);
27         FileInputFormat.setInputPaths(conf, new Path(args[0]));
28         FileOutputFormat.setOutputPath(conf, new Path(args[1]));
29         JobClient.runJob(conf);}}
```

Figure 5.3: Example of the Java skeleton code for the map/reduce in Hadoop framework.

There are two key elements that make this program skeleton specific to the Hadoop

framework. The program's classes are derived from the Hadoop framework class `MapReduceBase` and they implement the parameterized interfaces `Mapper` and `Reducer`. Also, the reader who is familiar with the Java types can see that the program contains new types such as `LongWritable`, `Text`, `TextInputFormat`, `TextOutputFormat`, and `IntWritable`. These types are introduced by the Hadoop framework as an alternative to Java types like `String` and `long` for a number of reasons. Objects of these types represent data chunks and they are serialized to a byte stream, so that they can be distributed over the network or persisted to permanent storage on the cluster computers. Distributing and persisting hundreds of thousands or millions of objects requires tight binary format and efficient implementation, something that the Hadoop framework provides.

In addition, these objects implement various interfaces that simplify map/reduce operations. For example, the interface `Comparable` exports methods for comparing data during key sorting in the reducer, and the interface `Writable` exports methods for saving objects to a local storage efficiently and without much of the overhead associated with the JVM's serialization framework.

After importing the Hadoop framework types in line `1`, the main implementation class `YourImplementationClassName` is declared in line `2` that contains the implementations of the mapper and the reducer, each of which are declared in the corresponding classes in line `3` and line `11` that extends the base Hadoop class, `MapReduceBase`. The mapper class, `YourMapperName` implements the method map of the interface `Mapper` that is parameterized by the types of the input and the output key-value pairs. In general, selection of these types depends on the semantics of the data, which in turns depends on the chosen conversion of the computation task to the map/reduce model. The parameter `reporter` passed in line `7` is used inside the method `map` in line `9` to send the progress of the mapper to the `ApplicationMaster`.

The mapper and the reducer implementation is missing in this example, since each map/reduce task can be encoded into this skeleton program. Both `map` and `reduce` methods are public member of their corresponding classes that are `static` meaning that their single objects exist and they can be accessed without instantiating their parent class `YourImplementationClassName`. Both methods output the results using the object `output` of the Hadoop `OutputCollector` type that is parameterized by the key-value types. The output is produced in line `10` and line `17` respectively. The implementation of the method `collect` uses the configuration of the Hadoop framework for a given task to write the results to the HDFS `DataNodes`. The Hadoop framework invokes the methods `map` and `reduce` automatically, it parses the input data and passes records to these methods as their input parameters. For those interested in in-depth implementation, the Apache Hadoop source code is publically available. [4].

The method `main` creates the job configuration object in like `19` and populates it with the information about the mappers and reducers in lines `20-26`. The classes are passed as the parameters to the corresponding methods of the configuration object

[4] `https://github.com/apache/hadoop-common/blob/1f2a21ff9b812ec9667ce7561f9c02a45dfe5674/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/lib/input/TextInputFormat.java#L42`

using the Java literal `class`. Recall from the *Java Language Specification (JLS)* that a class literal is an expression consisting of the name of a class, interface, array, or primitive type, or the pseudo-type `void`, followed by a dot and the token `class`. A class literal evaluates to the `Class` object for the named type (or for void) based on the class loader of the class of the current instance[5]. Finally, in lines `27-28` the input and the outpaths are set – they are assumed to be passed as command line parameters, and the job is run in line `29`.

5.8 Summary

In this chapter, we introduce the concept of big data and give examples of how implementation of the software to process big data is fundamentally different from the implementation of applications that process small amounts of data. We discuss the concept of the datacenter and show how it can be abstracted as a distributed object. Next, we show how the diffusing computation model that we reviewed before can be modified into the map/reduce model to split one task on the big data file into many thousands of map and reduce subtasks on much smaller files and how to implement and execute this model in a datacenter with failure management to allow users to view the execution in the datacenter as fault-free. We review the Google implementation of the map/reduce model, analyze new requirements for the filesystem, and then study how Google implemented these new requirements in its filesystem called GFS. We conclude by reviewing the case study of map/reduce implementation in Apache Hadoop.

[5] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.8.2`

Chapter 6

RPC Galore

Since most languages support a fundamental abstraction of encapsulating code inside a parameterized construct called a procedure, the RPC has been one of the most successful concepts that survived the test of time. RPC is used in multiple libraries, framework, and applications, and there are many various open-source and commercial RPC implementation. Initially, main effort focused on making RPC easy to use and incurring little overhead. Starting the mid 1990s the Internet enabled programmers to create distributed objects and host them on top of web servers, so that clients can access these distributed objects over the Internet. With the advent of big data, the importance of the RPC shifted towards making the underlying implementation efficient, easy to use, and having as little latency as possible, which is a major concern when millions of remote procedure calls are made by tens of thousand of clients per second. In this chapter we will review a number of popular RPC frameworks, compare them, and discuss their usages in big data projects.

6.1 Java Remote Method Invocation (RMI)

Recall from Chapter 4 that in RPC the platform and language neutrality is achieved by using the *Interface Definition Language (IDL)*, a common denominator language for creating specifications for distributed objects that can be accessed by clients written in different languages and running on different platforms, often not the same as the distributed object itself. A key to this approach is in applying an IDL compiler that takes the IDL specification as its input and generates the client and stub code automatically in different languages and for different platforms that can be distributed to the programmers to create the implementation of the clients and the distributed object. Therefore, the complexity of handling all language- and platform-specific implementation details is buried in the IDL compiler, which must be maintained and extended along with the languages and the platforms that it supports. Programmers concentrate on implementing applications' requirements and they do not need to worry about platform- and language-dependencies for the RPC.

The programming language Java is created based on the idea that its compiler gen-

erate the bytecode that runs on its own platform called the *Java Virtual Machine (JVM)*, which sits on top of the operating system but below the Java applications. That is, once written in Java, a program can run on any platform that hosts the JVM on top of which this program will run. In a way, Java becomes a common language denominator, like the IDL, except it allows programmers to create definitions of procedures in addition to their specifications or declarations. Therefore, there is no need to introduce a new IDL for Java RMI, since specifications of remote procedures can be created in Java. Of course, in real-world software projects Java is used alongside many other programming languages, and to enable interoperability between the JVM and many other languages and platform, Java IDL is used[1]. We will discuss it later in the book.

Recall that programmers create named interfaces in IDL with versions and GUIDs and the declarations of the procedures that they export, which include the names of the procedures, the sequences of the input and output parameters and their types. In Java, the notion of interface is a part of the *Java Language Specification (JLS)*: "an interface declaration introduces a new reference type whose members are classes, interfaces, constants, and methods. This type has no instance variables, and typically declares one or more abstract methods; otherwise unrelated classes can implement the interface by providing implementations for its abstract methods. Interfaces may not be directly instantiated[2]." This definition allows programmers to create declarations of the remote procedures and share them between clients and servers to enforce typing.

Until now we have not differentiated between procedural and object-oriented languages. In fact, we used the terms *function* and *procedure* interchangeably. One can think of a function as a mapping from the input values to the output values, whereas a procedure defines an ordered sequence of operations or commands that may not result in any output value. In Java, an object is a class instance or an array, where a class specifies a reference type (i.e., it is not one of the primitive types: boolean, byte, short, int, long, char, float, and double), where "types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong static typing helps detect errors at compile time[3]." Classes contain method definitions, which collectively define the behavior of the class instances. Invoking a method on an object is defined as `o.m(`$p_1, \ldots,$ p_n`)`, where `o` is the object name, and the parameters $p_1, \ldots, p_n$ are of types $t_1 \ldots, t_n$ respectively. A simple trick allows us to convert an object's method call into a template for the RPC – we extend the parameter space with the value of the object as the first parameter to the method call, i.e., `m(o_ref_value,` $p_1, \ldots,$ p_n`)`. Doing so enables us to reason about *object-oriented (OO)* remote method calls as the standard RPC, where the first parameter is used to resolve object references on the server side.

The behavior of Java RMI is defined in its specification[4]. It uses the client/server model, and one of the key element that distinguishes Java RMI from a plain vanilla RPC is the concept of a *registry*, a remote object that maps names to other remote objects[5].

[1] `http://docs.oracle.com/javase/7/docs/technotes/guides/idl/`
[2] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html`
[3] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.3.1`
[4] `https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html`
[5] `https://docs.oracle.com/javase/7/docs/platform/rmi/spec/`

Registries enable *transparency*, where any form of distributed system should hide its distributed nature from its users. The RPC provides *access transparency* as its basic benefit, where invocation of the procedure is the same regardless whether it is local or remote. The physical separation is hidden from the client. With registries, location or name transparency is enabled, where actual location of the remote procedure is unimportant and even unknown to the client, which uses unique object names, which are mapped and referred to by logical names in the registry. Later we will see how *migration transparency* allows remote objects to move and yet clients can access it during and after the move without changing the runtime configuration.

Since there are many tutorials, the reader can use them to learn how to program Java RMI on her own[6]. Here we will discuss briefly how using the JVM address some issues with the RPC such as parameter passing, lazily activating remote objects, and dynamically loading class definitions. But first, we will review the basic steps of the Java RMI process.

6.1.1 The Java RMI Process

The Java RMI development and deployment process starts with defining a remote interface whose methods represent remote procedures in the vanilla RPC. The remote interface extends the interface `java.rmi.Remote` and its methods should be defined as throwing `java.rmi.RemoteException`. Once defined, a programmer must create a class that implements this remote interface by defining its methods. The method `main` of the implementation class creates an instance of this class, exports a remote object using *Java Remote Method Protocol (JRMP)* [7], extracts the stub of this remote object, and binds a unique programmer-defined name to this stub in the RMI registry. The stub is returned by the `registry` to the client, so that it can be used as a reference to the remote object in the client program. These steps complete the barebone implementation of the Java RMI server program.

> **Question 1**: How does Java RMI framework guarantee that a single call to a remote method in the client program results in the corresponding single method call in the server program?

The implementation of the client program is straightforward. First, in the method `main` of the client class a reference to the `registry` is obtained. Next, a lookup is performed using the unique name that is assigned to the remote server object. The RMI framework returns the stub object that has the type of the remote interface. At this point, methods can be invoked on the stub object in the client program that will result in their corresponding server method invocations. Using remote interfaces enables compilers to statically type check if the signatures of the invoked methods match between the client and the server programs.

`rmi-registry2a.html`

[6] `https://docs.oracle.com/javase/tutorial/rmi/`

[7] `https://en.wikipedia.org/wiki/Java_Remote_Method_Protocol`

6.1.2 Parameter Passing

Passing parameters and returning computed values is a main technical issue to solve when designing and implementing an RPC mechanism. The main problem is to determine how types are represented on different platforms and in client and server programs, which are written in different languages. In Java RMI, serializable objects are all arguments to and return values from the methods of the remote objects. Java Object Serialization Specification specifies that objects that support `Serializable` interface can have their contents saved to a stream and then restored later from the stream to a new instance, and the stream includes sufficient information to restore the fields to a compatible version of the class[8]. Doing so enables Java RMI to use the JVM serialization as the effective mechanism for remote call (un)marshaling. Moreover, classes, for parameters or return values, that are not available locally can be retrieved at runtime by the RMI framework, as long as the client and server programs create the security manager object and enable permission to retrieve the code from the external server.

Recall that there are three main ways to pass parameters: by value, by reference, and by name. Passing by value means that the actual value of the variable is retrieved and copied onto the stack frame of the called method. If the size of the value is big (e.g., a long string) then it will take extra RAM to hold the copy of the value. Alternatively, passing by reference means that the address of the location in RAM is passed to the calling context, where the data is stored. Finally, passing by name enables programmers to pass complex expressions that will be re-evaluated every time they are used in the body of the method. These different ways of passing parameters have serious implications for the performance of the RPC in general.

> **Question 2**: How do you make a remote method call using Java RMI framework to pass a lambda function as a parameter to this remote method?

Consider passing a local object as a parameter to a remote method. If this object is fat, i.e., it contains a large amount of data in its fields, then it would make sense to pass this object by reference, since copying this data is expensive. However, address spaces are disjoint for the client and the server, therefore, the address of the object location in the client address space has no meaning for the server address space. As a result, the Java RMI specification states that a non-remote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, is passed by copy[9]. The RMI frameworks serializes the object to transmit it to the destination and unserializes it to construct the object and pass it as a parameter thus creating a new object in the calling JVM. Similarly, an exported remote object passed as a parameter or return value is replaced with its stub into the remote method. In addition, the RMI framework annotates serialized object with the URL of its class so that the class can be loaded at the destination JVM.

[8] `https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`

[9] `https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-objmodel7.html`

Question 3: How do you exclude certain fields from an object that you pass as a parameter to the remote method? That is, these fields can be used by the client program, but the server program will not "see" these fields when the object is received.

Consider a situation when there are multiple aliases in the client program to the same object, i.e., multiple variables are bound to the same object location. If the client and server programs are run in the same address space, then the object identity operation will establish that all these variables reference the object that has the same identity, i.e., the same object. However, if these variables are used as parameters to a remote method, the RMI framework will serialize and transmit these objects and then deserialize them into the server-bound objects that may have completely different identities even though they contain exactly the same data.

Question 4: What Java classes and methods are used to transmit parameters according to the Java RMI specification ?

To prevent this situation, the Java RMI specification establishes a requirement for the *referential integrity* that states that if two references to an object are passed from the client address space to a server address space in parameters or in the return value in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM.

Question 5: Assuming that you use only Java object serialization to pass objects between JVMs, how would you implement the referential integrity?

6.1.3 Lazy Activation of Remote Objects

In a large-scale deployment of a widely used Java RMI-based application, we can expect tens of million of distributed remote objects service millions of clients. Suppose that a remote object takes 100Kb of RAM, resulting in over 1Tb requirement for RAM to store 10Mil objects. Even though hardware gets cheaper and more affordable fast, the demand for successfull applications grows fast too, thus requiring more hardware to support. What's more, not all of these objects are required at the same time to service client requests, since in various contexts, client requests may come in bursts, where a period of the increased number of requests is followed by a period of lull. Also, the client demand fluctuates depending on the context, e.g., the demand for services may increase from 9AM to noon followed by the decrease during lunch time, resume again around 1:30PM, reach its peak around 4PM and drop after 5PM thus following a typical work schedule. To enable remote objects to become *active* when it receives requests from clients, i.e., to be instantiated and exported in a JVM, the Java RMI specification introduced the concept of *lazy activation*, where a remote object would not be

instantiated and exported in a JVM (i.e., *passive*) until the first request arrives from a client. Doing so enables an automatic conservation of computing resources until they are required.

A key idea behind the innerworkings of the lazy activation is to encapsulate a reference to a remote object's stub in a fault object that contains an activation identifier and the actual reference to the remote object that may be `null` if the remote object is passive on the first call. The first time a client invokes a method of the remote passive object, the fault is issued and then the RMI framework loads and initializes the remote object and performs a method call. The reader should read the RMI specification for in-depth discussion of the activation protocol.

6.2 <your data serialization format here>-RPC

An important part of an RPC implementation is the definition of the data exchange format. Back in the 1980s and a good part of the 1990s the formats were binary, i.e., precisely defined ordered sequences of bits identified semantic elements of a message. Both the client and the server adhered to a chosen protocol based on a fixed data format. Since the RPC is supposed to bridge differences across multiple languages and platforms, a binary format must be flexible to add new data types and various tags that designate how to convert data. For example, consider the issue with little and big endian representation where the former designates the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address, whereas the latter does the opposite. Intel is historically based on little-endian format and Apple computers use big-endian format; GIF images use little-endian and JPEG images use big-endian. A data format must account for all attributes of data representation and the (un)marshalling components of the RPC must implement a chosen data format to ensure that data conversion is done properly. Examples of widely used binary data formats are *Network Data Representation (NDR)* in Distributed Computing Environment (DCE)[10] and eXternal Data Reprentation (XDR)[11].

In general, conversions of data in the RPC can be *asymetric*, where the client program must have complete information how the server represent data or vice versa where the server must know the clients' representations of the data. That is, if one client uses the NDR and the other client uses the XDR formats, the server must use corresponding data converters for these formats. Of course, if both clients and the server use the same data format, the runtime overhead is small and the conversion is straightforward. Alternatively, to reduce the dependency on the knowledge about the used data format, a *symmeric* data conversion approach uses some intermediate representation to which calls from the client and responses from the server are converted. One example of a format that can be used as an intermediate representation is Abstract *Syntax Notation One (ASN.1)*, where each data item is represented with a tag that defines the type of the data item, its length as the number of bytes in the field value, and the value of the data item. We will refer to such encoding as a *type-length-value (TLV)*. ASN.1 is very

[10] `http://pubs.opengroup.org/onlinepubs/9629399/chap14.htm`
[11] `https://docs.oracle.com/cd/E19683-01/816-1435/6m7rrfn9n/index.html`

influential, it is used in many protocols and it heavily influenced data formats in Google RPC and Apache/FB Thrift that we will consider below.

6.2.1 XML-RPC

Since the introduction of the *eXtensible Markup Language (XML)* in 1996, it has been used widely to encode documents and define their schemas in a human- and machine-readable format, its performance was acceptable for many applications and it was easy to use [8]. XML has become a de facto lingua-franca of data interchange in electronic commerce. The success of XML is dictated by the pervasiveness of its predecessor, *HyperText Markup Language (HTML)*, a markup language that is used to create web pages for display in Internet browsers. For example, an HTML page that lists components that exist on a computer is shown in Figure 6.1.

```
1 <html> <head>
2 <title>Computer with IP address 192.168.1.1</title>
3 </head>
4 <body><h1 align="center"><font size="7">Components</font></h1>
5 <ol><li><p align="left"><b>Realplayer</b></li>
6 <li><p align="left"><b>IE Web Browser</b></li></ol>
7 </body></html>
```

Figure 6.1: A fragment of HTML code describing a list of items on a computer.

HTML contains tags whose job is to tell a web browser how to display the web page. HTML offers a set of tags that are instruc- tions given to web browsers on how to format text. For example, the tag `<h1 align=center>` specifies that a browser should use the top heading style and align the displayed text to the center of the browser display area. Unfortunately, this specification format does not carry any semantic meaning about the data. When it is displayed in the browser a reader can extract the semantics of names and map it to the basic concepts.

The solution of this problem lies in XML's markup tags that allow developers to design custom types that map to predefined concepts and structure data semantically. For example, the HTML data shown in the previous example are given in XML format in Figure 6.3.

```
1 <components> <computer Address="192.168.1.1">
2 <component>Realplayer</component>
3 <component>IE Web Browser</component>
4 </computer></components>
```

Figure 6.2: A fragment of XML code describing a list of items on a computer.

XML documents are organized in a hierarchical way as trees where the root node specifies the topmost parent XML element and children elements are defined under their respective parents. Each element has zero or more attributes, which are key-value pairs. The basic unit in XML is element and XML enables declaration of data using custom types that are instantiated as elements. Each element may contain attributes that provide additional information. For example, element `computer` has the attribute

`Address` with value `192.168.1.1`. There is a plethora of XML parsers on the market, some of them embedded in language runtimes and incorporanted as part of the type systems, e.g., Scala.

Just as language grammar serves as a specification that dictates the rules of writing programs XML uses schemas and *Document Type Definitions (DTDs)* to enforce certain structure of XML documents. DTDs specify which elements are permitted in XML documents that use these DTDs. For example, a DTD for the XML document that describes components is given below.

```
1 <!ELEMENT components (computer*)>
2 <!ELEMENT computer (component+)>
3 <!ATTLIST computer Address CDATA #REQUIRED>
4 <!ELEMENT component (#PCDATA)>
```

Figure 6.3: A fragment of XML code describing a list of items on a computer.

This DTD declares that element components may have zero or more child elements called computer. Each computer element may have one or more elements called component. As in regular expressions an expression followed by sign + means one or more repetitions, and followed by sign `*`, i.e., the Kleene star[12] means zero or more repetitions. Element computer has attribute `Address` that is declared as string data `(CDATA)` and this attribute is required. Finally, element component is declared as parsed character data `(PCDATA)`, and it means that this element contains some text. This DTD can be included in the XML document by adding the following line: `<!DOCTYPE components SYSTEM components.dtd>`. `DOCTYPE` is a markup telling the XML parser that a DTD is included, `components` is the name of the top-level element and the name of the DTD in quotes. `SYSTEM` is an URI that is used to describe the scope of the specification.

Programmers can easily create custom types, which map to semantic concepts of the system that they model. We give a small example of the implementation of a calculator with one method `add` that returns the sum of two integer parameters.

The Java-like pseudocode example is shown in Figure 6.4 of the XML-RPC-based implementation of the calculator with the single method `add`. Lines 1–11 show an example of the actual XML request sent by the client over the network to the server. The XML header in lines 2–7 defines the protocol with which this request will be transmitted (i.e., *HyperText Transfer Protocol (HTTP)*), the method `POST` with which this request will be applied to the destination web server `www.somehost.com`, among other things, and the payload that includes the root element `methodCall` that contains the children elements `methodName` and `params` that define the class and the method names and paramters values and types. The client code that emits this request is shown in lines 22–31. In the method `main` of the client class `CalcClient` the object of the XML-RPC framework class `XmlRpcClient` is instantiated whose constructor takes the address of the webserver that hosts the XML-RPC server. The parameters are added to the container class, `Vector` in lines 26–28. Finally, the call `execute` is invoked on the object in line 29 where it takes the name of the server class and the

[12] `https://en.wikipedia.org/wiki/Kleene_star`

```
1  //XML-RPC request
2  POST /XMLRPC HTTP/1.1
3  User-Agent: UserName/15.2.1
4  Host: www.somehost.com
5  Content-Type: text/xml
6  Content-length: 500
7  <?xml version="1.0"?>
8  <methodCall>  <methodName>Calculator.add</methodName>
9        <params><param> <value><i4>3</i4></value> </param>
10              <param> <value><i4>7</i4></value> </param></params>
11 </methodCall>
12 //XML-RPC Server
13 public class XMLRPCServer {
14 public Hashtable add(int x, int y) {
15         Hashtable result = new Hashtable();
16         result.put("add", new Integer(x + y));
17         return( result );}
18 public static void main( String [] args ) {
19         WebServer server = new WebServer(8080);
20         server.addHandler("Calculator", new XMLRPCServer()); }
21 //XML-RPC Client
22 public class CalcClient {
23         private String server = "http://www.someserver.com";
24         public static void main (String [] args) {
25                 XmlRpcClient client = new XmlRpcClient( server );
26                 Vector params = new Vector();
27                 params.addElement(new Integer(3));
28                 params.addElement(new Integer(7));
29                 Hashtable result = (Hashtable) client.execute (
30                                 "Calculator.add", params );
31                 int sum = ((Integer)result.get("add")).intValue();  } }
```

Figure 6.4: Java-like pseudocode example of the XML-RPC-based implementation of the calculator with the single method `add`.

mehod and its parameters and returns a hash table that contains the result, which is retrieved in line 31.

The XML-RPC server class is shown in lines 13–20. It contains the implementation of the method `add` in lines 14–17 that puts the sum of two integers into the hash table object `result` and returns this table object. In the method `main` in lines 18–20, the webserver is started and the instance of the class `XMLRPCServer` is registered with the webserver, under the key `Calculator` thus activating it and making it available to clients to call its method remotely.

Last, one can think of combining XML with Java RMI to access remote Java objects over the Internet. This was the idea behind the *Java Application Programming Interface (API)* for *XML-based RPC (JAX-RPC)*. Eventually, it was renamed into *JAX for Web Services (JAX-WS)* and we will mention it later in the book. The JAX-RPC service is based on the W3C (World Wide Web Consortium) standards like *Web Service*

Description Language (WSDL)[13].

6.2.2 JSON-RPC

Recall the *JavaScript Object Notation (JSON)*, a widely used lightweight and language-independent data interchange format was introduced just before the end of the last millenium[14]. A key element of the JSON representation is a text-based key-value pair, where the key specifies the name of an attribute whose value is followed after the separator colon, e.g., `"jsonrpc": "2.0", "method": "add", "params": [3, 7]`. In this example, we have comma-separated key-value pairs, where `jsonrpc` is the key that designates an RPC call with the value `2.0` designating the version of the JSON-RPC framework, the key `method` specifies the name of the remote method, and the key `params` specifies the value that is an array designated with the square brackets that contain two parameter values. Submitting this JSON data object to the destination webserver that hosts the JSON-RPC framework will cause the invocation of the remote method and the result of this invocation will be passed to the client using the JSON object, e.g., `"jsonrpc": "2.0", "result": 10`.

Question 6: Discuss the advantages and drawbacks of YAML[15] Ain't Markup Language-RPC compared with JSON-RPC.

From these examples the reader can see that JSON-RPC is quite similar to the XML-RPC, it is stateless, and it can use multiple message-passing mechanisms to transmit the message. The JSON-RPC specification allows batching several request objects to reduce the number of round trip communications between the clients and the server[16]. To date, there are hundreds of various implementations of JSON-RPC with various degrees of use.

Since remote RPC-based objects can be deployed on top of webservers, one may submit RPC calls to the remote methods from the command line. A popular utility for doing that is `cURL`, a command line tool and library for transfering data with URL syntax and it supports multiple protocols and various encoding and security options[17]. Consider the following command: `curl --user drmark --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getinfo", "params": [] }' -H 'content-type: text/plain;' http://bitserver` that submits a JSON request to a bitcoin server to execute a remote method `getinfo`[18]. There are multiple implementation on the Internet where the `cURL` library is used to implement client API calls for various RPC frameworks.

[13] `https://github.com/javaee/jax-rpc-ri`
[14] `http://www.json.org`
[16] `http://www.jsonrpc.org/specification`
[17] `https://github.com/curl/curl`
[18] `https://en.bitcoin.it/wiki`

6.2.3 GWT-RPC

Google Web Toolkit (GWT) is an Apache-licensed open source set of tools that allows web programmers to create and maintain *Asynchronous JavaScript And XML (AJAX)* applications using the Java language[19][20]. Since GWT applications are based on the client/server model, where the client is a *Graphical User Interface (GUI)* that allows users to interact with the back-end of the GWT application that is usually hosted in the cloud. The GUI may be hosted in a browser or can be run as a desktop application on a user's computer and it communicates with the server using some networking protocol. The user interacts with the GUI by sending events to its GUI objects (e.g., entering data into a text box or clicking on a button) and the GUI client responds to these events by executing code that communicates with the server, which performs computations and may generate and return a new GUI (e.g., a *HyperText Markup Language (HTML)* page generated from a Java-based distributed object). In GUI applications, user interactions often lead to multiple invocations of remote methods in the back end.

Since GWT applications frequently fetch data from the server, the RPC framework is integrated in GWT to exchange Java objects over HTTP between the client GUIs and the back-end servers[21]. The RPC server component is implemented as a *servlet*, which is a Java class that exposes well-defined interfaces according to the Java Servlet Specification, and servlets are deployed in web containers, i.e., webservers that receive HTTP-based client requests to invoke remote methods exposed by the deployed servlets [22]. The GWT framework exports classes and interfaces similarly to the Java RMI, where the interface `RemoteService` must be extended by a created remote interface that contains declarations of the exported remote methods. A service implementation must extend the GWT framework class `RemoteServiceServlet` and must implement the service's interface. Once compiled, it can be deployed with the webserver as a servlet, making it available to clients. The latter uses the method `create` of the class `GWT` to obtain a reference to the remote object and bind the reference to a specific endpoint (i.e., the URL) where the remote object is deployed by casting to the interface `ServiceDefTarget` and invoking its method `setServiceEntryPoint` with the URL as its parameter. This simple procedure underscores the ubiquity of the RPC that is used in many different frameworks. Interested readers may look into Errai, a GWT-based framework for building rich web applications based on the RPC infrastructure with uniform, asynchronous messaging across the client and server[23].

[19] `http://www.gwtproject.org`

[20] `https://en.wikipedia.org/wiki/Google_Web_Toolkit`

[21] `http://www.gwtapps.com/doc/html/com.google.gwt.doc.DeveloperGuide.RemoteProcedureCalls.html`

[22] `https://jcp.org/en/jsr/detail?id=315`

[23] `http://docs.jboss.org/errai/latest/errai/reference/html_single/#sid-5931313`

6.3 Facebook/Apache Thrift

Facebook is a social media company with over one billion active daily users who post and read various news items in a multitude of formats[24]. Loading a single user's homepage involves hundreds of servers at its internal datacenters, making remote calls to retrive tens of thousands of data objects with strict time constraints that usually do not exceed one second of response time[25]. Facebook uses hundreds of thousands of commodity computers in its datacenters[26], as reflected in its financial filings that it spent over $3.63 billion in 2016 for equipment. As for software, Facebook uses dozens of open-source frameworks and programming languages including `MySQL`, `Hadoop`, `Cassandra`, `Hive`, `FlashCache`, `Varnish`, `PHP`, `Python`, `C++`, `Java`, and functional languages `Erlang`, `OCaml`, and `Haskell`. A reason that Facebook engineers created their own RPC design is that other existing RPC systems had multiple drawbacks: heavyweight with significant runtime overhead (e.g., XML-based, CORBA), proprietary (e.g., gRPC), platform-specific (Microsoft RPC), missing certain features (e.g., Pillars misses versioning), or a having less than elegant abstraction.

Facebook Thrift is an RPC framework conceived, designed, and implemented at Facebook and released to the Apache open source [141]. In 2010, Thrift became an *Apache Top Level Project (TLP)*. Thrifts goal is to provide reliable communication and data serialization across multiple languages and platforms as efficiently and seamlessly with the highest performance. Multiple Thrift tutorials are available on the Internet[27]. Here we discuss main elements of Thrift's design and implementation.

Thrift's design revolves around five concepts: to enable programmers to use native types, to decouple the transport layer from the code generation, to separate data structures from their representation when transported between clients and servers, and to simplify the versioning when rolling out changes to clients and servers frequently.

Types. Thrift IDL offers a type system that allows programmers to annotate variables with types that map to the native types of the destination languages and platforms. Base types include `bool`, `byte`, and bit-designated types like `i16`–`i64` for integers whose representation lengths are defined by the corresponding number of bits. Types `double` and `string` are correspondingly a `64`-bit floating point number and a text/binary sequence of bytes.

Question 7: Why doesn't Thrift IDL contain unsigned integer types?

Composite types in Thrift IDL include structures where fields are annotated with unique integer identifiers thus keeping their track across multiple versions, three types of containers: `list`, `set`, and `map`, exceptions, which are equivalent to structures, and services, which contain method declaration with annotated parameters and return types, using the basic types, exceptions, and structures.

[24] `http://www.facebook.com`
[25] `http://www.datacenterknowledge.com/the-facebook-data-center-faq/`
[26] `https://techcrunch.com/gallery/a-look-inside-facebooks-data-center`
[27] `https://thrift.apache.org/tutorial`

Transport. In general, programmers should not think about how objects are exchanged between clients and servers. Once they are serialized, the choice of the communication mechanism should not matter and it should not affect how they are converted into an internal representation as sequences of bytes. Thrift provides the interface `TTransport` that exports abstract methods `open`, `close`, `read`, `write`, `flush`, and `isOpen` – they designate generic operations that can be performed on all kinds of communication mechanisms. Concrete implementations of this interface include `TSocket` and `TFileTransport` and programmers can choose them based on the semantics of the RPC implementation.

Protocol. In Thrift, the encoding of objects as messages does not matter between clients and servers. All data types in Thrift are encoded into binary messages with a high degree space optimization using the network byte order. All strings are prepended with integers that designate their length, and all complex types are decomposed into their integer constituents. Only field identifier attributes are used at destinations, the names of the fields do not matter. Method names are sent as strings.

Versioning. Versioning is done by analyzing field unique identifiers and determining what fields are modified and which ones are added between the versions.

Remote dynamic dispatch is implemented using generated maps of strings (method signatures) to function pointers. Thrift compiler is implemented in C++ using lex/yacc. C++ forces explicit definitions of language constructs. Thrift is used in many services including Facebook search and logging service and in many other companies around the world. Readers interested in Thrift can download and run it and read the original documentation and the paper on Thrift to learn more details [141].

6.4 Google RPC (gRPC)

Google created its RPC (gRPC) implementation to enable programmers to quickly build remote distributed objects in a variety of languages seamlessly. A key innovation in gRPC is *protocol buffers (protobufs)*, a language- and platform-neutral, extensible fast and efficient mechanism for serializing structured data. The authors consulted a few companies and organizations and he observed that protobufs are used not only in gRPC but also to prototype data structures and interfaces of the distributed objects as part of writing requirements specification before software is even designed. Google created plenty of documentation and tutorials for gRPC and they are publicly available with the implementation of gRPC[28].

In this section, we concentrate on two aspects of gRPC: the protobuf implementation and the load balancing. Defining the structure of messages and services is similar to Thrift, where each message field is assigned a unique order numeral tag and integer types are defined with the number of bits that are used in the destination language to represent values defined by these integers. Field unique numeral tags are used to identify fields in the message binary format, which helps versioning. Once defined, fields

[28] `http://www.grpc.io/`

cannot be assigned to a different tag or a tag cannot be reassigned to a different field if the previous one to which this tag was assigned to, is deleted. Nesting is allowed and maps container types are added to the protobuf language. However, the most important idea is that the `protoc` compiler takes the protobuf message definition input and generates the code in the desired language that constructs, serializes, and deserializes the protobuf messages. The generated a class uses the pattern `Builder`, which exports methods for all protobuf messages and their fields, and messages can be constructed by instantiating this class and calling methods for each fields with concrete values to construct the entire message and emit it in the internal wire-specific format. Conversely, a constructor takes a constructed binary protobuf and parses it to populate the object fields that programmers can access directly. Multiple converters exist to translate protobuf messages into JSON, XML, and many other formats, and vice versa.

Similar to Thrift, gRPC is built as highly performant in the distributed datacenter environment. Since there are many clients requesting services from gRPC remote objects, a load balance, which runs in the process `grpclb`, distributes requests from gRPC clients to balance the load across multiple available remote objects. The workflow starts with a gRPC client that issues a name resolution request as part of making a remote call. A response indicates either *Interner Protocol (IP)* address of the load balancer process or a configuration service address that describes what policy to choose to distribute clients' requests (e.g., the first available address or a round robin selection)[29]. If the returned address belongs to the load balancer, the gRPC framework will open a stream connection to a remote object server that is selected based on the algorithm that calculates a projected distribution of workloads.

6.5 Twitter Finagle

Twitter is a company whose business is to enable its registered users to write short messages asynchronously and read messages posted by other users. As of the middle of 2017, Twitter has over 100 million daily users that make over 500 million remote calls to post their messages, 20% of which contain images. Highly asynchronous nature of communications between the users and the Twitter platform influenced the design of the Twitter RPC framework called *Finagle* [47], which is an open-source and used in many software projects outside Twitter[30]. Finagle is written in Scala, a functional language built on top of the JVM. Finagle's design is rooted in three abstractions: *composable futures* to stitch together results of asynchronous operations, *asynchronous and uniform services* with standardized API specifications, and *connecting filters* that allow programmers to compose separate services into larger modules. A main idea is to enable Finagle's programmer to specify what they want to accomplish rather than code low-level details of how to accomplish the desired goal, step-by-step.

Futures are automatic waits on values, when reference to a value is unknown at creation of binding, and this reference will automatically be filled up by some computation. That is, a future represents an abstraction for an object to hold some

[29] `https://github.com/grpc/grpc/blob/master/doc/service_config.md`
[30] `https://twitter.github.io/finagle`

value that is not available when the object is created, but it will be available after some later computation. Before the computation occurred, the corresponding future is empty, if the computation resulted in an exception, the future will fail, or it will succeed if the computation produces a value.

```
1 val numberOfFollowers: Future[List[String]] = Future {
2       connect2TwitterAccount(credentials).getFollowers().toList }
3 val numSentMsgs = numberOfFollowers.filter(_.status == PERSONAL).
4     map(followerObject => SendMessage(followerObject, someMessage)
5 numSentMsgs onSuccess {case => println("sent "+numSentMsgs.toString)}
```

Figure 6.5: Scala pseudocode example of using futures to send messages to Twitter followers.

Consider Scala pseudocode for sending a message to all followers of some Twitter account that is shown in Figure 6.5. The variable, `numberOfFollowers` declared in line 1 receives the list of the Twitter account names of all followings of some account defined by `credentials`. Of course, it is needless to say that the method `connect2TwitterAccount` may take some time to execute and the client program does not have to block and wait until this method finishes. It continues to execute the code to line 3, where it uses the variable, `numberOfFollowers` to filter out all accounts that are not personal (e.g., corporate) and then to send messages by mapping each account oject to the method `SendMessage` in line 4. Finally, in line 5, the method `onSuccess` is defined to print out the list accounts to which messages were sent successfully. So the meaning of `future` is to enable programmers to concisely defined asynchronous calls without writing additional code that defines how these asynchronous calls can be composed in a sophisticated pipelines with error handling.

Asynchronous uniform service example is shown in Figure 6.6. Importing classes from the Finagle framework is done in line 1 and the server class is declared in line 2. The service object is created in line 3 as an instance of the parameterized class `Service` whose method `apply` takes the single HTTP request parameter in line 3 and returns the future of the HTTP response. The implementation of the method `apply` returns an HTTP response with the version number and the status `Ok`. The service is bound and exported in line 6

```
1 import com.twitter.finagle.*
2 object Server extends App {
3    val service = new Service[http.Request, http.Response] {
4    def apply(req: http.Request): Future[http.Response] =
5         Future.value(http.Response(req.version, http.Status.Ok))}
6    Await.ready( Http.serve(":8080", service) ) }
```

Figure 6.6: Scala pseudocode example of a Finagle's remote service.

This example comes from Finagle's tutorial[31] and we use it to illustrate how easy it is to create asyncronous services and export their remote methods with Finagle.

[31] `https://twitter.github.io/finagle/guide/Quickstart.html`

In fact, Finagle can be used with other RPC services like Thrift for which Twitter developed and released Scrooge[32], a Thrift code generator written in Scala that makes it easy to integrate Thrift and Finagle RPCs[33].

Connecting filters enable chaining of Finagle services into a composite service and they are defined in Scala as functions `type Filter[Req, Resp]=(Req, Service[Req, Resp]) => Future[Resp]`. That is, a connecting filter takes the request as its parameter and applies the function `Service` to it to obtain a response that is returned as `Future`. For example, consider the identity filter `val identityFilter = (req, service)=> service(req)` and the timeout filter `def timeoutFilter(d: Duration) = (req, service) => service(req).within(d)`. Combining them will yield the following pipeline using the combinator `andThen`: `timeoutFilter andThen identityFilter andThen client`.

6.6 Facebook Wangle

Similar to Finagle, *Facebook Wangle* is a client/server RPC framework for building asynchronous event-driven high-performant pipelined C++ services[34]. An event is a data structure or a record that is created asynchronously and placed in memorywhere it can be accessed using some IPC mechanism. The reader will find many similarities with Finagle by analyzing the semantics of the pipelining. In Wangle, *handlers* are programs that produce and consume events and handlers are connected via the event dataflow thus enabling programmers to compose handlers into pipelines. Composability and scalability are two main primary goals of Wangle, where two building blocks are used: futures and folly/io/async. The latter , a special library. Folly/io/async is a set of the OO asynchronous I/O wrappers around *libevent* and it provides event bases, sockets, and asynchronous timeout/callback interfaces[35]. Libevent is a widely used high-performant library for specifying and executing callbacks in asynchronous calls[36]. Interested readers can find a wealth of documentation and tutorials on the `code.facebook.com` website.

6.7 Summary

In this chapter, we reviewed some of the most popular and most used RPC frameworks. The powerful idea of invoking remote methods over the network has become even more important in the age of cloud computing. Modern implementations of RPC use more powerful abstractions like futures and pipelining to make it easy for programmers to create highly performant RPC services without writing much code. Making a single remote call or a sequence of synchronous remote calls is easy and well-understood;

[32] `https://twitter.github.io/scrooge`
[33] `https://twitter.github.io/scrooge/Finagle.html`
[34] `https://github.com/facebook/wangle`
[35] `https://github.com/facebook/folly/tree/master/folly/io/async`
[36] `http://libevent.org`

constructing a large pipeline of asynchronous remote calls where remote objects hosted on different platforms are accessed by hundreds of millions clients billion times a day presents a significant challenge. The main idea of this chapter is to build a picture of the complex world of software constructed from distributed objects using the mosaic of the RPC frameworks.

Chapter 7

Cloud Virtualization

The meaning of the word "virtual" is having the essence or effect but not the appearance or form of, according to the British Dictionary. With respect to computers, virtual <X> means that <X> is not physically existing as such but made by software to appear to do so. Virtualization of computer resources is easy to illustrate using the concept of the *virtual memory*, which is much larger volatile storage than the available physical memory. Of course, if a computer has only 4Gb of the physical memory and the virtual memory is configured to be 10Gb, it does not mean that it would be possible to load and manipulate 10Gb of data into the memory at once. Instead, the *memory management unit (MMU)* of the OS relies on the assumption that all 10Gb of data will not be needed at the same time, but instead only a few smaller chunks of the data will be needed at a time. Therefore, the user will have a view of contiguous address space with the capacity of 10Gb, whereas in reality the MMU will load and remove pages with data on demand, a technique known as paging. This example points out to an important property of a virtualization in general – the presence of some resource in a virtual world is an abstraction, which is realized by some processes that execute instructions that implement the specified behavior of this resource.

In this chapter, we will discuss how the concept of resource virtualization enables streamlined deployment of applications in the cloud. After examining the benefits of the virtualization abstraction for enabling automatic (de)provisioning of resources to applications, we will review the processes of simulation of computing environments and emulation of the execution of applications compiled for one platform on different platforms. Finally, we will study the organization of virtual machines and analyze popular virtualization solutions used in various cloud computing environments.

7.1 Abstracting Resources

Consider a remote object accessed by their clients using the RPC. Now, suppose that this remote object simulates a computer and its operating system. As part of this simulation, it is possible to pass some other object as a parameter to a method of the remote object and then expose the interfaces and their methods of the passed object from within

the remote object. A client can then invoke the exposed methods of the passed objects, which will be executed by the remote object that simulates the computing environment. Going even further, we can take one remote object that simulates a computer and pass it using the RPC as a parameter to a method of some other remote object that also simulates a computer, so that one simulated computer will run into some other simulated computer. And we can continue this composition of remote objects ad infinitum.

Viewing a computing environment as an object with exposed interfaces is an example of a resource abstraction. A computing resource is a hardware or software unit that provides services to its clients who may be humans or other resources. Basic resources are atomic units like the CPU or a memory cell. Composite resources consist of basic and composite resources that exchange data by invoking one another's interface methods. By abstracting resources we view them as closed objects with exposed interfaces that define the objects' behavior by invoking methods of these exposed interfaces according to some protocol. Since the objects are closed, we do not expose their implementations as a part of the abstraction – only their interfaces are important. The CPU can be viewed as an object whose interface allows clients to call methods for submitting bytecode for execution, loading and parsing the next instruction, executing the loaded instruction, and other relevant operations. Viewing a resource as an abstract object enables designers to provide a uniform view of the system's architecture even if the implementation of these resources can change from time to time.

Question 1: Please discuss how quickly a running application can use a newly provisioned vCPU.

To illustrate the last point, consider a datacenter with 100 physical CPUs – we chose this number simply for convenience. By monitoring the execution of the applications, suppose we observe that many CPUs are underutilized due to a different reasons: applications block on I/O, sleep on synchronization, or simply because the number of executing threads in applications is smaller than the number of CPUs. Now, suppose we implement and instantiate 1,000 *virtual CPUs (vCPUs)* as software objects that we described above that take the instruction sets of the applications and execute these instructions. Of course, implementing the instruction loop as it is shown in Figure 2.2 is not enough – a physical CPU is needed to execute both the vCPU loop and the instructions. However, by placing a vCPU monitor between the actual CPUs and vCPUs allows us to schedule instructions for vCPUs on the actual CPUs that leads to satisfying some load objectives. For example, we can allow customers to acquire 200 vCPUs for a single application even though the physical system has only 100 actual CPUs. If the vCPU monitor does a good job of scheduling unused CPU time to assign it to the customer, she can never know that there are more vCPUs allocated to her applications than there are physically available. And in that lies the power of abstracting resources.

7.2 Resource Virtualization

Recall the meaning of the word "virtual" as having the essence or effect but not the appearance or form of. An abstraction of the resource presents its projection that leaves some of its properties out – abstracting the CPU means that we do not consider some of its interfaces and its internal structure, e.g., the implementation of the operation that loads the instruction and its operands into the memory. However, when virtualizing the CPU, its implementation can reveal more details to its users than the actual physical CPU, e.g., internal memory locations that implement registers can be made available to the users. In fact, an implementation of a vCPU can be much more complex than the architecture of the physical CPU which executes the instructions of the vCPU. Thus, virtualized resources can either simulate real resources or emulate them.

Suppose that a virtualized computer presents virtual resources that do not exist to users, hence the virtualized computer *simulates* a desired computer environment even if it may exist only in a specification and not in concrete hardware, i.e., simulators imitate the behavior of the system. For example, pilots are trained on simulators, which do not create the real place that lifts a pilot above the ground and moves her in space at the designated speed. Similarly, a virtual computer can simulate access to a network site that does not exist. Virtual resources may have more complex behavior, e.g., distributed computing objects may enable programmers to stop the components of the running process across multiple computing objects, undo actions of some instructions, skip some instructions, and resume the process. Simulating this behavior involves the actual hardware resources like networking, CPUs, and RAM on physical computers, but there may not be any direct mapping between the instruction to undo actions of some instructions of the running process on a virtual computer and concrete CPU instructions that accomplish the undo operation.

Alternatively, *emulators* bridge the gap between the designs of the source and the target system often by translating one set of instructions into some other instruction set that is accepted for execution by the target system. Consider a Unix terminal emulator that runs on Windows and it takes Unix instructions and translates them to the corresponding Windows instructions. That is, the Unix terminal is an original Unix program that executes on top of an Windows emulator. A virtual resource simulator may use emulation to translate the instructions of this virtual resource to the target system.

> **Question 2**: Please discuss how emulation is related to program compilation.

The most significant drawback of resource virtualization is the worsened performance, since the software implementation is almost always slower than the direct use of the physical resource. Moreover, the implementation of a virtual resource may contain bugs – a study in 2015 reported 117 confirmed bugs that make vCPUs violate the CPU specifications [10]. Additional problems include weakened security, where virtual resources may reveal more information from the applications that use them compared to the actual resources, and the need for monitoring virtual resources and scheduling them, leading to the additional overhead on physical resources.

Despite all these problems virtualization of resources offers multiple benefits [35].

Whereas it is very difficult to change hardware resources to add or remove some functions, it is relatively easy to accomplish with virtual resources. Moreover, it becomes possible to install new operating systems that are written for a different hardware architecture on top of a virtual architecture that is realized using virtual resources. Also, it makes possible to capture the state of program execution with the states of all virtual resources and even replay and modify the execution of the program as needed. Doing so is not limited to a local execution, but also to distributed executions where component programs are executed in different (virtual) address spaces and communicate via (virtual) network, making it possible to synchronize the states of distributed virtual resources. Last, but not least, a uniform security policy can be applied to virtual resources making it possible to apply fine-grain security policy on the resource level.

Question 3: A *honeypot* is a computer security mechanism set to protect computer systems from an unauthorized use by creating a deceptive computer environment that appears real to the attacker. Is a honeypot a simulation or an emulation of a real computing environment? Please elaborate your answer.

7.3 Virtual Machines

A *virtual machine (VM)* is a virtual programmable computer that is designed and implemented as an incremental modification of a lower level machine called its base [69]. One can view the Turing machine as the lowest level base-0, however, realistically, a real computer hardware assembled according to some computer architecture specification is considered to be the lowest base-0 machine, which often includes the hardware memory, the CPUs, the GPUs, the I/O architecture that comprises the hard disk storage with controllers, network cards, and controllers for peripheral devices. Applications can run on top of base-0, or more often, on top of the OS, whose purpose is to present a higher level abstraction on top of pure hardware base-0 layer. Applications can access and manipulate hardware resources in base-0 via system call interfaces. The reader can view system calls as the RPC or vice versa, where a separate address space is assigned to base-0 and the OS exports interfaces for system calls to allow applications that run in a different user address space to invoke remote methods from the address space of base-0 to interact with hardware resources. Generally, VMs are stacked in hierarchical structure where the lower VM provides some services and the higher level VM offers some abstraction of these services to its higher-level VM.

There are many popular commercial and open-source VMs, such as the *Java Virtual Machine (JVM)*, the .Net *Common Language Runtime (CLR)*, the Android RunTime (ART), the *Low Level Virtual Machine (LLVM)*, and *Node.js*. All VMs are defined in their respective specifications that outline the memory organization, the instruction set, the formats of executable files, and data types among many other things. For example, the JVM specification states that the JVM is an abstract computing machine that has an instruction set and manipulates various memory areas at runtime. Interestingly, the CLR and the JVM specification do not explicitly require the JVMs to run on top of

operating systems, so the VMs can be implemented on top of computer hardware manipulating it directly. However, the specifications do not state how hardware resources are accessed and manipulated, e.g., how they are shared across multiple programs that use these resources. These VMs are often referred to as high-level language VMs, which are a subset of the category of process VMs, which translate a set of OS and user-level instructions written for one platform to another [142].

Alternatively, a system VM provides a complete computing environment with virtual resources that emulate the underlying hardware and a hosted guest OS. In some cases, a system VM can obtain information about the underlying hardware and create virtual resources that mimic the computing environment it is installed on. Doing so allows multiple VMs to run in the cloud environment efficiently and the underlying *VM monitor (VMM)* can schedule VMs to run on different commodity computers even migrating them to improve load balancing as the VMs continue to execute applications hosted inside of them.

Question 4: Can a process VM host a system VM? What about the other way around? Can a system VM host another system VM?

Different types of virtualization are used to create VMs. In QEMU, Bochs, and PearPC, emulation is used to translate the instruction set from the VM to the destination lower base. Fully virtualized machines host operating systems that run applications as if these OSes are directly installed on top of the underlying computer hardware. In the extreme, a fully virtualized machine can be installed on top of the guest OS that is already hosted inside some other fully virtualized machine – however, it is unlikely to be useful from a practical point of view.

In two other types of virtualization, the operating system is adjusted to work with virtual resources to deliver VMs [144]. In *paravirtualization*, the virtual hardware resources differ from their actual counterparts and the VM is designed for the modified virtual resources; however the OS is modified too to run as a guest OS inside the VM to address the changed virtual resources. This is why paravirtualization is also called OS-assisted virtualization. For example, consider that a vCPU does not export some instruction that the OS relies on. This instruction will be replaced in the OS by calls to an exported method of the VMM called a *hypercall*, which will simulate the behavior of the CPU that the OS expects. Doing so often results in higher performance of the VM, since the overhead of emulation can be avoided.

An example of using paravirtualization is how Xen handled the problem with the x86 architecture that requires that programs that need to control peripheral devices write a control data sequence to a specially allocated I/O memory space. Each device is mapped to a specific address range in this I/O memory space. The hardware controller reads the data sequence from the I/O memory space and sends signals to the physical device via the bus. Of course, the I/O memory space is treated by the OS differently from other memory locations, e.g., it is not paged or cached. And this memory space is linked directly to the hardware via a controller, making it more difficult to virtualize.

Question 5: Explain how you would virtualize peripheral resources in the x86 architecture with full virtualization and emulation.

The other interesting aspect of the x86 architecture is that it has four privilege levels known as ring0 to ring3, where user applications run in ring 3 and the OS must run in ring0, since only the latter allows the instructions to directly access the memory and hardware. Since the VM runs in the user space and the guest OS runs within the VM, it runs in ring3 and it would not function as an OS to the user applications that run on top of it. Applying full virtualization requires the emulation of all OS instructions by the VMM to translate them into their semantic equivalents accepted by the ring x86 architecture. Full virtualization may reduce the performance of the OS by 20% or more, whereas paravirtualization, as intrusive as it is, results in a better performance.

On a side note, hardware manufacturers took notice of a rapid expansion of virtualization and offered enhancements to the underlying hardware architectures to enable seamless access to virtual resources to improve performance and reduce the amount of work that VM producers must accomplish to implement full virtualization. The idea of *hardware-assisted virtualization* or *accelerated virtualization* rests on extensions to baseline processor architectures, so that the VMM can use instructions from these extensions directly thereby avoiding modifications to the guest OSes. Consider the Intel's hardware virtualization technology (Intel VT), where, among many things, additional instructions are introduced that start with letters "VM" [1]. For example, with the instruction VMLAUNCH, the VMM can run a specific VM and with the instruction VMRESUME it can resume execution of a VM. Using hardware-assisted virtualization yields the benefits of not having to change the OS kernel, however, it costs some additional complexity in the CPU design and runtime overhead, leading to a hybrid approach where paravirtualization is combined with accelerated virtualization.

In paenevirtualization, the internal VMs are created by the OS and they run within the OS as containerized user processes where containers are isolated from one another by the OS kernel. A prominent example of paenevirtualization is *Linux-VServer* where the user-space environment is partitioned into disjoint address spaces called *Virtual Private Servers (VPS)*, so that each VPS is an isolated process that behaves as the single OS kernel to the user-level processes that it hosts [2]. Each VPS creates its own context to abstract away all OS entities (e.g., processes, resources) outside of its scope and strictly control and monitor interactions between contexts and processes that run within these contexts [3]. Different levels of isolation are applied to resources, where files in shared directories are less isolated than network sockets, shared memory, and other IPC mechanisms.

Paenevirtualization are often referred to as a variant of the *chroot jail*, a runtime environment for a process and its children with a changed root directory thus isolating processes and resources that are visible to them. Consider FreeBSD, an open-source

[1] `https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html`

[2] `http://linux-vserver.org`

[3] `http://linux-vserver.org/Paper`

popular OS that was released in 1993 [96]. As part of its distribution, FreeBSD contains a VMM called `bhyve` that can host guest OSes that include Windows, BSD distributions, and Linux flavors. Moreover, VirtualBox and QEMU emulator run on top of FreeBSD. In addition, paenevirtualization is implemented in FreeBSD called *jail lightweight virtualization*, where a *jail* is a process space that contains a group of processes and it has its own root administration with full isolation from the rest of the OS. Access control is used in jails to prevent accessing address spaces of the other jails, namespaces are created for each jail to give the impression of a fully global namespace environment where name collisions are avoided with other jails, and `chroot` is used to constraint the jail to a subset of the filesystem. Processes that run within jails are confined to operations that can access addresses and resources that are bound to their respective jails where these proceeses run. For example, jailed processes cannot reboot the system or change network configurations, and these and other restrictions may limit the use of paenevirtualization depending on the needs of the hosted applications.

Despite all restrictions, paenevirtualization offers a number of benefits: low overhead since neither emulation nor software-based resource virtualization are used, no need to installed virtualization software that requires special virtualization images for simulated computing environments, and no need to install and configure guest OSes. These benefits are rooted in the resource isolation principle that is also a drawback, since resources are not virtualized, and it means that it is not possible to move VMs or cluster them to utilize resources in the datacenter efficiently. In addition, the OS kernel must be modified, and this is an intellectually intensive and error-prone exercise.

7.4 Hypervisors

A *hypervisor* is a VMM that controls the execution of VMs and it supplies them with virtual resources [122]. As such, a hypervisor is a software process that runs between the underlying platform and the actual VM that it hosts. Native (or type-1 or bare-metal) hypervisors run directly on top of computer hardware, whereas hosted or type-2 hypervisors run on top of OSes. Hybrid hypervisors combine the elements of type-1 and type-2 hypervisors, e.g., the Kernel Virtual Machine (KVM*)* that is embedded in the Linux kernel and it runs on top of accelerated virtualization hardware thus acting both as an OS kernel and the type-1 hypervisor that runs KVM guests that run applications on top of embedded vCPUs and device drivers.

The theory of hypervisors was formulated in 1974 as the Popek and Goldberg virtualization requirements [122], which are based on the notion of VMM map that is shown in Figure 7.1. The entire state space is partitioned into two sets: the states that are under the VMM control, S_{VMM}, on the left, and all other states, S_O, on the right of the VMM map. A program, P, is a sequence of instructions, each of them maps a state to some other state. Suppose that there is a program, P, that transform some state, S_i into the state, S_j in

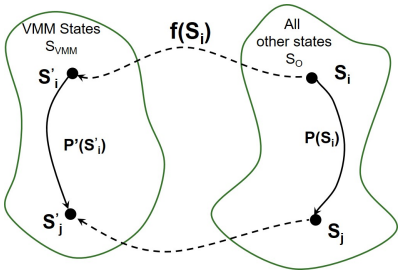

Figure 7.1: VMM map.

the partition, S_O as it is shown in Figure 7.1 with the solid arrows. A VMM map is a structure-preserving map, $f : S_O \rightarrow S_{VMM}$ between the partitions S_O and S_{VMM} that establishes a condition that if there is a set of instructions, P, in the state partition S_O that maps some state S_i to some other state, S_j, then there must be a corresponding set of instructions, $P^{'}$ in the state partition S_{VMM} that maps the corresponding state $S_i^{'}$ to some other corresponding state, $S_j^{'}$, which are linked by dashed arrows.

The VMM map leads to formulating the following Popek-Goldberg fundamental properties that determine relations between states and instructions.

Performance property dictates that all instructions that do not require special treatment by the hypervisor must be executed by the hardware directly. This property stipulates that the performance of the VMs that the hypervisor hosts should reach in the limit the performance of the native execution environment without the hypervisor by allowing a range of instructions to bypass emulation.

Safety property prohibits an arbitrary set of instructions that are not executed by the hypervisor to control resources. This property establishes a level of protection between the VM and the hypervisor that guarantee that no rogue instruction can take control of the resources that the hypervisor manages.

Equivalence property states that results of the execution of a set of instructions on by the hypervisor are indistinguishable from the results of the execution of the same instruction set without a hypervisor on the same platform. This property establishes a level of protection between the OS and the applications inside the VM, ensuring that the results of the execution of the application within the VM are the same for the same application in the native, i.e., the hypervisor-free environment.

Given that a hypervisor should satisfy these properties, what is required to create a hypervisor for a given hardware platform? This question is answered in the following theorem formulated by Popek and Goldberg: a hypervisor can be created if and only if the union of control-and behavior-sensitive instructions is the subset of the set of all privileged instructions. The proof of the theorem is by construction, i.e., we will show how to create a hypervisor for a hardware platform.

Question 6: Does the Popek-Goldberg theorem apply to process VMs?

To explain the meaning of control-and behavior-sensitive instructions, we make certain assumptions about a hardware platform and the OS that can run on this platform and will be hosted by the guest VM that will run on top of the hypervisor. The platform and the OS support the kernel and the user modes and switching between them is accomplished using traps (exceptions), which are system calls that are implicitly invoked by the OS to handle certain situations like dividing by 0 or accessing a NULL memory address. Traps differ from interrupts, which are caused by peripheral devices, e.g., keyboard events, or clock ticks. Since interrupts are dealt with by firmware and not by user processes, we ignore them here.

In the user mode, the CPU executes non-privileged instructions in programs, e.g., arithmetic instructions. Once a privileged instruction is detected, e.g., an I/O operation or resetting the CPU or memory address mapping, the CPU switches to the kernel mode, where the CPU saves the information necessary to continue the execution of the user process and it passes the control to the OS, a process that must run in the kernel mode as dictated by the underlying hardware architecture. Assuming that memory accesses do not result in any traps, executing privileged instructions in the user mode results in non-memory traps whereas executing them in the kernel mode will not result in the same traps. Nonprivileged instructions can execute in both modes without traps.

A simplified model of the OS for the VMM is based on two concepts only: the supervisor/user mode and the notion of virtual memory, the schema of which is shown in Figure 7.2.

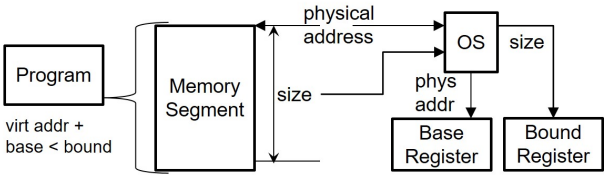

Figure 7.2: Virtual memory schematic organization.

Executing a privileged instruction results in a trap if the program is in the user mode and no trap is triggered if the program is in the kernel mode. When it comes to the virtual memory, the OS manipulates the base and bound registers to store the physical address of the memory segment assigned to a program and the size of this memory segment, respectively. Traps are triggered when a memory location accessed by the program violate the bound. We are interested in instructions that can manipulate the state of the CPU or the registers for the virtual memory in the user mode without traps.

Here comes a dilemma – an OS is designed to run in the kernel mode, but it runs in the VM in the user mode, and when it attempts to execute privileged instructions, the execution results in a trap. It means that to handle this case, the hypervisor must receive the privileged instructions from the OS and to process them. This is the essence of the *trap-and-emulate architecture* where the hypervisor runs the VMs directly and the VMM handles traps and emulates privileged instructions.

Next issue is with the use of the virtual memory, where the OS maps virtual address space to the CPU's physical address space using the MMU. A hardware mechanism to implement virtual memory is by using specialized hardware called relocation and bound registers. In a crudely simplified description, the CPU generates some address, A that is compared with the address in the bound register, B. If $A \geq B$ then the system traps, otherwise the address from the relocation register, R that keeps the location of the program segment and it is used to compute the physical address, $A+R$. Both bound and relocation registers are read and written into only by the OS using special instructions.

We call an instruction *control-sensitive* if it can read or write the control state of the architecture, specifically, the kernel/user mode bit and the virtual memory registers. Thus, executing control-sensitive instructions gives the information about the environment in which the program is running and changes the environment to affect the execution of all programs in it. Example of control sensitive instruction includes an instruction that instructs the OS to return to the user mode without triggering a trap. *Behavior-sensitive* instructions are those whose semantics vary depending on the settings of the control registers without accessing them directly (e.g., returning physical

memory address given some virtual address and it depends on the value of the relocation register). Example of behavior-sensitive instruction includes loading a physical memory address or performing actions that depend on the current mode. Regular or innocuous instructions are neither control nor behavior sensitive.

Now, we have enough information to show how to construct a hypervisor. If the control-sensitive instruction can read the kernel/user mode bit and this instruction is not privileged, then running this instruction in the guest OS within the VM will result in a different answer compared to running this instruction outside the VMM, thus violating the equivalence property. For the behavioral-sensitive non-privileged instruction of returning the physical address for a given virtual address, the location register is different in the OS versus the hardware one, thus also leading to the violation of the equivalence property.

A hypervisor, $H \triangleq < \mathcal{D}, \mathcal{A}, \Xi >$, where $\mathcal{D}$ is its top control module called a dispatcher that decides which other modules to load. $\mathcal{D}$ should be placed in the location where the hardware traps, meaning that whenever a privileged instruction traps, $\mathcal{D}$ will respond. After $\mathcal{D}$ determines what resources are requested, it passes the control to the module $\mathcal{A}$ that allocates resources for the VM that requested it. Finally, Ξ is the set of interpreter routines that are invoked by $\mathcal{A}$ in response to privileged instructions. This triple defines a hypervisor and let us consider the cases.

Suppose that the hypervisor is in kernel mode, and all VMs it hosts are in the user mode. The hypervisor handles all traps and all transitions from the kernel to the user mode using its module $\mathcal{D}$. When an application makes an OS system call, the call is a trap that is intercepted by the hypervisor that decodes this call and transfers the control to the OS running in the VM under the application that made the system call.

The hypervisor virtualizes the physical memory. Specifically, the VMs occupy the physical memory that is virtualized by the OS as a virtual memory for the applications that run inside the VM, whereas the hypervisor controls the actual physical hardware memory. The memory in the VM is called the *guest physical memory* and the hardware memory is called the *host physical memory*. VM's relocation and bound registers are virtualized and controlled in the VM by the OS, whereas their physical counterparts are controlled by the hypervisor, whose module $\mathcal{A}$ maps the registers in the implementation using offsets and values. Summarily, the hypervisor, H handles all privileged instructions that conform to the Popek-Goldberg properties and if the control- and behavior-sensitive instructions are a subset of the privileged instructions, then H is the hypervisor program that conforms to the Popek-Goldberg properties.

Question 7: Explain if recursive virtualization is possible under the Popek-Goldberg properties, where a VM runs a copy of itself using the hypervisor.

7.5 Interceptors, Interrupts, Hypercalls, Hyper-V

Generally, an *interceptor* is a collective term for an approach and techniques to replace an instruction invoked by a client with some instruction set that may include the original instruction without the knowledge of the client while having complete access to

the context of the instruction call. In the context of VMs, when an application that resides within a VM makes a call to access some privileged guest OS instructions, the hypervisor intercepts this call and performs other calls on the behalf of the OS kernels. These interceptors located in the hypervisor are hypercalls. In general, one can think of an interceptor as a wrapper around some code fragment (e.g., a library, a method, or an instruction) that presents the interfaces, methods, and instructions with the same signatures as the wrapped code fragment. Within the interceptor/wrapper, additional functionality may monitor the interactions between the client and the wrapped code fragment or it can completely replace the wrapped instructions with some new code. Interceptor can be added at runtime without any disruption to clients' interactions with the wrapped server code using various binary rewriting techniques.

Interrupts are signals that are produced by hardware or software components to inform the CPU that events happened with some level of importance. The CPU may discard the signal or respond to it by interrupting the execution of the current process and executing a function called *interrupt handler* in response to this signal. When a physical device (e.g., a keyboard) raises an interrupt, the signal is delivered to the CPU using some architecture-specific bus and the CPU saves the existing execution context and switches to executing the corresponding interrupt handler. Software interrupts (e.g., division by zero) may be handled by the OS or the interrupt can be propagated to the CPU. Some interrupts must be processed extremely fast with low overhead (i.e., clock interrupts) and some software and hardware components generate many interrupts (e.g., a network card).

Consider the virtualized platform where a guest OS runs in a VM that is executed by vCPUs. If the guest OS or a virtualized hardware component raise an interrupt, it should be delivered to the vCPU, however, a vCPU is also a software component that is scheduled by the hypervisor to execute on the physical CPU. Since the hypervisor multiplex virtual components to execute on the physical CPU, when a virtual device raises an interrupt, the hypervisor may switch to assign a different virtual device process to the CPU, and since no relevant vCPU is running, interrupt processing is deferred until the vCPU is executing. Doing so may have a serious negative effect on the responsiveness of the virtualized interrupt handling mechanism.

Consider how the TCP/IP networking stack processes network card interrupts that require the execution of a large number of CPU instruction. For data-intensive applications that process network I/O from external data sources (e.g., sensors, mobile phones), it is important that the processing time is minimal. However, virtualized networking devices and vCPUs introduce delays due to waiting for the hypervisor to assign these virtualized devices to the physical CPU. As a result, guest VMs that handle a large amount of network traffic show significantly worsened performance when compared to the native hardware devices. One solution is to attach network devices directly to the guest VMs to improve the performance, however, doing so defeats the purpose of sharing physical devices among multiple applications.

In this section, we discuss how the challenges of interrupt handling are implemented in the architecture of Hyper-V, a hypervisor created by Microsoft that can run virtual machines on x86 CPUs running Windows. Its architecture is shown in Figure 7.3. The dashed line shows the separation between the kernel and the user spaces and the hypervisor type 1 runs on top of bare hardware.

A key notion in Hyper-V is a *partition*, defined as an address space with isolation enforcement by the hypervisor to allow independent execution of guest OSes. The hypervisor creates partitions and assigns a guest OS to run in each partition except for the root partition that controls all work partitions. Work partitions do not allow the guest OSes to access physical hardware and process interrupts, these work partitions present virtual devices to processes that run on guest OSes.

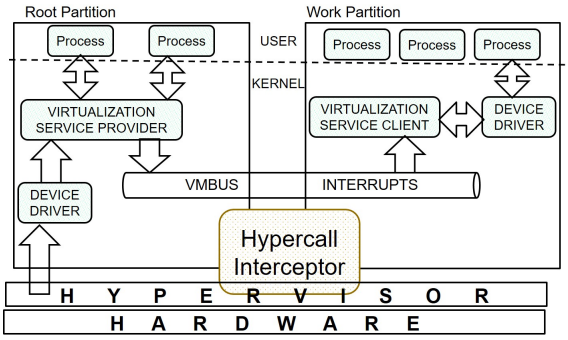

Figure 7.3: Hyper-V architecture.

Inside the root partition, *Virtualization Service Providers (VSPs)* receive and process requests from virtual devices. During processing, the VSPs translate requests into instructions that are performed on physical devices via device drivers that are running within the root partition. The VSPs communicate with the VM worker processes that act as proxies for the virtual hardware that is running within the guest OS. Interrupts and virtual device messages are transfered using the VMBus between work partitions and the root partition. VMBus can be viewed as a special shared memory between the root and work partitions to provide a high bandwidth and low latency transfer path for the guest VMs to send requests and receive interrupts and data items. Specific VMBus messaging protocols enable fast data transfer between the root and work partition memories, so that data in work partitions can be referenced directly from the root partition. Using these techniques improves the response time between physical devices and virtual machines.

7.6 Lock Holder Preemption

Virtualization introduces a range of problems, one of which is called *lock holder preemption problem (LHPP)* [54, 150]. It shows the conflict between well-established concepts and their implementations in OSes and how they affect virtual environments. A key problem with LHP is with the spinlock, a syncronization primitive that enables a lock-acquiring thread to wait actively on this lock by monitoring it in a loop. Conversely, blocking or sleeping locks block the waiting thread until the lock is released, however, it is a solution that has a number of problems. First, a blocked thread may not respond in time to certain interrupts. Second, when a thread is blocked, the OS puts it to sleep thus saving and eventually restoring its context, which are very expensive operations, especially if a thread will wait on a lock for a short time. This is why spinlocks are widely used in many OSes.

Now, recall that a vCPU is a software application that executes program's instructions in a loop. The hypervisor schedules the execution of a vCPU on physical CPUs

for some periods of time called time slices. It works the same way as scheduling threads to execute user programs, where the code of the vCPU is executed during a time slice in a thread, then the thread is preempted, its context is saved, the context of the other thread is restored and the execution continues.

Consider what happens when a vCPU executes a thread that obtained a spinlock. When its time slice expired, the thread that executes the vCPU that executes the thread that holds a spinlock is preempted. The spinlock is not released, since the hypervisor cannot force the vCPUs release spinlocks arbitrarily, otherwise, it may lead to all kinds of concurrency problems. Since the spinlock is being held by the preempted thread, other running threads will waste their cycles waiting on the spinlock until the context switch happens again and the code that the vCPU runs will finally release the spinlock. This is the essense of the LHPP.

Performance impact of LHPP varies, it was shown that in some cases over 99% of time the vCPUs are spinning thereby wasting the physical CPU cycles [54]. One solution to LHPP is to replace spinlocks with blocking locks, as it was done in OS^v that we will review in Section 8.3. Other solution includes the use of I-Spinlock, where a thread is allowed to acquire a lock if and only if the remaining time-slice of its vCPU is sufficient to enter and leave the critical section [148]. There are many ways to eliminate spinlocks by using lock-free algorithms that depend on hardware atomic instructions or by modifying the hypervisor to exchange information with the guest VM to avoid wasted spinlock cycles by allowing the vCPU that holds the spinlock to finish the computation and release the lock, and then to preempt that thread. It is still an ongoing research to create algorithms and techniques to solve the LHPP.

7.7 Virtual Networks

So far, we have discussed hardware virtualization in general, but we gave example of virtualizing internal resources of single computers like the CPU and the RAM, however, it is possible to extend the notion of virtualization to *local area networks (LANs)* where computers are connected in a local environment (e.g., an office) and *wide area networks (WANs)*, which connect two or more LANs (e.g., the Internet). A common abstraction of a physical network is a graph that supports network services where nodes represent VMs and edges represent tunnels that implement connectors using which VMs can send and receive messages. Popular *Open System Interconnection (OSI)* model abstacts network services into seven layers (also called a network stack) to implement protocols in seven layers and it was created by the International Standards Organization (ISO).

L7 is where programs exchange data using some protocol (e.g., emailing or transferring files using HTTP, FTP, SSH);

L6 handles data translation from some program format into a network format (e.g., encrypting messages or representing images as text);

L5 controls connections between computers as sessions, which are temporal logical boundaries that deliniate message exchanges in a network. Stateful sessions

keep the history of communications where requests and responses depend on the previous history, whereas in stateless sessions each request/response pair is indendepent from the previous history;

L4 ensures that messages can be disassembled into variable-length sequences of bits and delivered correctly with error recovery and flow-control algorithms in WANs;

L3 message transfers between multiple nodes in a LAN;

L2 establishes data links between computing nodes directly;

L1 deals with physical aspects of data exchange using electrical signals as bits.

Throughout this book we will refer to these layers to discuss various abstractions and solutions. For example, when we discussed the RPC client/server model, we referred to L7 as the application layer where a client program sent requests to the server programs without discussing the details of networking. When we expanded the discussion to stubs, we referred to as L6 to discuss networking formats such as NDR and XDR for remote call encoding. When discussing pipelining in various RPC implementations, we moved to L5 to discuss session control among collections of services on the network. L5-L7 are often referred to as application layers, L4 as the TCP/IP protocol, L3 as a message routing layer that also involves TCP/IP protocol, L2 as switching, and L1 as physical signal transfer. L1-L2 are also referred to as Ethernet, a term that encapsulates a range of networking technologies that include both hardware and message protocols defined in IEEE standard 802.3.

In many ways, these layers represent different abstractions of various services by concentrating only on their essential elements. However, one common thing that these layers have is that the control and the data are bundled together, where bits in messages can represent both application data (e.g., parameters of the remote methods) and control actions (e.g., the IP address of a VM to route a message to). Separating data and control planes is a key idea in the virtual or *software defined network (SDN)*.

Question 8: Discuss pros and cons of the "intelligent routing" where L3 collects information about all possible routes and selects an optimal one to send a message to the destination node.

Before we discuss SDNs, let us recall the elements of network organization. The Internet backbone is the collection of many large WANs that are owned and controlled by different commercial companies, educational institutions, and various organizations, which are connected by high-speed cables, frequently, fiber optic trunk lines, where each trunk line comprises many optic cables for redundancy. A failure of many WANs in the Internet backbone will likely not lead to the failure of the Internet, since other remaining WANs will reroute messages that would otherwise be lost. Other networks include smaller WANs and LANs: residential, enterprise, cellular, small business networks. These networks are connected to the Internet backbone using routers, which are hardware devices for the network traffic control by receiving and forwarding messages. Bridges are hardware devices that connect network segments on L1-L2 levels.

Whereas routers allow connected networks remain independent using internally different protocols while sending messages to one another, bridges aggregate two or more networks that use the same protocol using hardware addresses of the devices on the network. Routers and bridges forward the network traffic indiscriminately, and to manage the flow of data across a network, hubs/switches are used to forward a received message only to the network nodes to which the message is directed. A graph where nodes represent networking devices and WANs/LANs and edges represent physical wires connecting these nodes, is called a *network topology*.

An SDN is a collection of virtual networking resources that allow stakeholders to create and configure network topologies without changing existing hardware networking resources by separating the control and the data planes. The SDN controller reads and monitors the network state and events using programmatic if-then rules, where antecedents define predicates (e.g., `FTPport == 21`) and consequents define actions when antecedent predicates evaluate to true (e.g., `discard(HTTP_POST_message)`). A typical firewall rule can discard messages that originate from IP addresses outside the LAN. Subsets of the network traffic can be regulated by separate custom-written controllers.

General design goals for SDNs include flexibility of changing network topologies, manageability for defining separate network policies and mechanisms for enforcing them, scalability for maximizing the number of co-existing SDNs, security and isolation of the networks and virtual resources, programmability, and heterogeneity of networks. An SDN resides on top of different routers, switches, and bridges that connect various LANs and WANs, and a programmer can define virtual devices and connections among them to redefine the underlying network into a new topology with rules for processing messages based on the content.

Question 9: Describe load balancing mechanism for an RPC-based application using an SDN.

In a multitenant cloud datacenter, each computer hosts multiple VMs, and uneven changing workloads require different network topologies. A network hypervisor for an SDN is used to create abstractions for VM tenants: with the control abstraction, tenants can define a set of logical data plane elements that they can control; with the packet abstraction, data is sent by endpoints should see the same service as in a native network. To implement these abstractions, the SDN hypervisor sets up tunnels between host hypervisors. The physical network sees only IP messages and the SDN controller configures the hosts's virtual paths. Logical data paths are implemented on the sending hosts where tunnel endpoints are virtual switches and the controller modifies the flow table entries and sets up channels.

Data plane contains streaming algorithms that act on messages. Routers get data messages, check their headers to determine the destination. Then, the router looks up the forwarding table for output interfaces, modifies the message header if needed (e.g., TTL, IP checksum), and passes the message to the appropriate output interface.

7.8 Programming VMs as Distributed Objects

Exposing a hypervisor as a programming server object in the RPC style enables its clients to invoke the remote methods of this object. In this section, we use VirtualBox Software Development Kit (SDK) documentation to illustrate the programmatic interfaces of this open-source hypervisor for x86 architecture that is maintained by Oracle Corporation[4]. Even though some API calls are specific to VirtualBox, they can easily translate into other API calls exported by other hypervisors (e.g., vmWare and Xen), since the underlying concept of the VM organization remain largely the same. The interface `VirtualBoxManager` represents a hypervisor that exports the interface `IVirtualBox` for managing virtual machines. Its methods allow clients to obtain the list of VMs that are hosted by the hypervisor or a specific VM using its identifier (e.g., the name). A specific VM is representated by the interface `IMachine` and its components (e.g., hard drive, vCPU) are obtained using methods of the interface `IMedium`. That is, the organization of the SDK follows the hierarchical organization of the simulated computing environment. A wealth of the source code examples can be found in the public SVN repository for VirtualBox[5].

```
1  import org.virtualbox_5_1.*;
2  VirtualBoxManager mgr = VirtualBoxManager.createInstance(null);
3  mgr.connect("http://url:port", "uname", "pswd");
4  IVirtualBox vbox = vboxManager.getVBox();
5  IMachine vmFrom = vbox.findMachine(VMNAMEFROM);
6  ISession sesFrom = mgr.getSessionObject();
7  vmFrom.lockMachine(sesFrom, LockType.Write);
8  IMachine vmTo = vbox.findMachine("VMNAMETO");
9  ISession sesTo = vbox.getSessionObject();
10 vmTo.lockMachine(sesTo, LockType.Write);
11 IMedium from=sesFrom.getMachine().getMedium("DVNM", port, dev);
12 sesFrom.getMachine().detachDevice("DEVNAME", port, dev);
13 sesTo.getMachine().attachDevice("DVNME", port, dev,
14         DeviceType.HardDisk, from);
15 sesFrom.getMachine().saveSettings();
16 sesTo.getMachine().saveSettings();
17 sesFrom.unlockMachine();
18 sesTo.unlockMachine();
```

Figure 7.4: Java pseudocode for moving a hard disk from one VM to another VM in VirtualBox.

Consider the Java pseudocode example for moving a hard disk from one VM to another VM in VirtualBox as it is shown in Figure 7.4. The pseudocode does not include any exception handling and error management, and its goal is to illustrate what can be accomplished with the basic VirtualBox SDK[6]. The framework package is imported in line 1 for the version of VirtualBox `5.1`. In line 2, the object of the class `VirtualBoxManager` is created and it is an abstraction for the hypervisor that hides

[4] `http://download.virtualbox.org/virtualbox/SDKRef.pdf`
[5] `https://www.virtualbox.org/svn/vbox/trunk/`
[6] `https://www.virtualbox.org/sdkref`

low-level virtualization details. In line 3, the client connects to a specific computer that hosts the hypervisor and in line 4 it obtains the object of the interface `IVirtualBox`. In line 5 and line 8, references are obtained for VMs from which the hard drive is take and to which this drive is attached. In lines 6-7, a session object is created to obtain a lock on the VM from which the drive is taken, and the same is accomplished is lines 9-10 for the VM to which this drive is attached. Next, the object of the interface `IMedium` is obtained in line 11 and it serves as a reference for the hard drive that is used to move the drive between the VMs in lines 12–16. The VMs are unlocked in lines 17-18. Overall, this procedure is referred to as *hot swap*, since the VMs can execute their guest OSes that run user programs. The reader can use her imagination to construct various computing environments automatically according to some declarative specifications that can serve as inputs to clients of VMs.

7.9 Summary

In this chapter, we describe the theory of virtualization in the cloud. We explained the basic abstractions that enable us to represent hardware resources as abstract entities with interfaces that can be implemented in software. After reviewing the concepts of simulation and emulation of computer environments, we discuss the process and system *virtual machines (VMs)* and explain how they are hosted by the VM monitor that exports virtualized hardware resources. Next, we present the Popek-Goldberg theory of hypervisors, explain the fundamental properties of virtualization, and give the Popek-Goldberg theorem with its proof that directs how to build a hypervisor. Then, we extend the concept of virtualization to computer networks, and we briefly discuss virtual or software defined networks. We conclude this section by showing that VMs can be manipulated programmatically by clients as distributed objects where clients can manipulate virtual resources, start and shutdown VMs and programs that run in them, and combine them in a pipelined execution.

Chapter 8

Software Appliances for Cloud Computing

Creating software applications using third-party applications – libraries, components, or objects – has always been a difficult exercise. Producers of these third-party applications faced a daunting and arduous task of ensuring that they test their software under a variety of environmental factors that their users can encounter on target computers, such as the version of the operating systems, drivers and packages installed on that OS that may conflict with the software applications, and the values of environmental variables that may be conflicting or simply incorrect. As a result, the cost of third-party applications was much higher to pay for customer support, troubleshooting, and often sending qualified personnel to the customers' sites to help them install and configure purchased software applications.

An analogy with hardware appliances tells us that many installation and maintenance difficulties can and should be avoided. Purchasing a refrigerator and installing it in a residence is a routine exercise that does not require sophisticated skills and a significant time and resource investment. Modern refrigerators, microware oven, washers/dryers and many other appliances are connected to the Internet and its manufacturers can update software running on these hardware appliances remotely and even connect to them to troubleshoot. Consider modern digital multimedia refrigerators with a touch screen in a 15.1-inch thin-film-transistor liquid-crystal display (TFT-LCD), a LAN port, the built-in MP3 player, an electronic temperature control system with the ability to monitor food content and expiration dates. The border is blurred between pure kitchen appliances and strictly computer hardware appliances, which traditionally included routers, game consoles, and printers among others.

The appliance analogy can be extended to view a software application as a composition of distributed objects each of which is deployed in a customized VM. These preconfigured VMs that are ready to run on hypervisors are called *virtual appliances (VAP)* [129, 130]. A cottage industry is created around building, selling, or exchanging VAPs, where software developers create VAPs with preconfigured software stacks, VAP game servers, or specialized VAPs that provide ranges of prices for specific items.

For example, Citrix Netscaler, an expensive and traditionally hardware-implemented load balancer is currently sold as a VAP that provides, among many things, data compression and caching[1]. If a programmer needs an object that provides a certain service, all she needs to do is to find a VAP that provides this service, download and install the VM on the cloud infrastructure, and it is ready to provide services by exposing its interfaces with remote methods for client programs[2].

Question 1: Take a commodity workstation and minimize its architecture by removing hardware and replacing it with VAPs. Describe the minimal hardware design of the workstation.

When we mention configuring a VAP, one component that we have treated as immutable up to this point is the guest OS. However, the hypervisor assumes a few functions of an OS such as context switching between running processes, (de)provisioning memory, and handling filesystems. One can wonder why a fully loaded OS is required for a VAP that runs a distributed object, which may need only a fraction of OS services. In fact, why should a VM emulate the entire computing environment? In many cases, the sound cards, PCI busses, floppy drives, keyboard, bluetooth and other hardware are not needed to deploy distributed objects. Moreover, the more virtual hardware is enabled in the VM, the higher the risk of some security attack that may exploit this hardware. For example, in VENOM attack, a flaw in the virtual floppy disk controller in the emulator QEMU enabled a denial of service attack by using out-of-bounds write and guest crashes and even executing arbitrary code[3]. Thus, by optimizing the OS to include only those components that are needed *just enough* to run a software application, it is possible to reduce its size, make it faster by removing redundant instructions that consume the CPU time, make it efficient, since it will consume fewer resources, and more secure. Combining this *Just Enough OS (JeOS)* – pronounced as the word "juice" – as a guest OS in a VM with a software application for which this JeOS is configured is called creating a *software appliance*, a concept that supersedes the notion of VAP. Software appliances can be connected in a *virtual cluster*, where each VAP communicates via a virtual Network Interface Controller (vNICs), a device that connects a computing node to the network by receiving and transmitting message from and to routers at L1-L4 OSI layers, and virtual routers with other applications hosted in VMs over the WAN. Virtual clusters, in turn, can be connected in *virtual datacenters*, which we will discuss later in the book.

8.1 Unikernels and Just Enough Operating System

Type-1 hypervisors take a few functions away from the OS: provisioning resources, enabling virtual network communications, and context-switching when running multiple

[1] zzzzzzzzzzzsssshttps://docs.citrix.com/en-us/netscaler/10-5/vpx/ns-vpx-overview-wrapper-con.html

[2] https://marketplace.vmware.com/vsx

[3] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456

VMs by multiplexing on a shared cluster of physical commodity computers in a datacenter. In fact, by working with LBs, VMs can be relocated when running applications to execute on a hypervisor whose hardware resources are underutilized[4]. However, even though fewer functions are required from guest OSes, they are still installed and configured using their full distribution packages as if they control the entire physical computer. Moreover, full OS distributions result in bloated VM image files: Microsoft Windows takes around 20GB[5], and that installation comes with a minimum number of packages. Reports show that the full installation may grow to 100Gb. On the other hand, selecting only the base packages for different flavors of the guest Linux OS may reduce the VM image to less than one gigabyte. Thus, a problem is how to define the minimum subset of the guest OS services that are required for applications to run on top of this OS.

Question 2: What OS services are required only for sending and receiving messages as sequences of bytes?

Three important insights show that there is a way to slim down the guest OS image. First insight is that there may not be any need for multitasking in a guest OS. If a VM is instantiated to run only one application, then the OS can be slimmed down by removing the code that keeps track of processes and switches between them. Since it is a single process that is executed by the guest OS, there is no need for protected disjoint address spaces and keeping track of offsets for running process to ensure that they do not access an address space that belongs to some other processes. These simplifications of the guest OS remove unnecessary code, make its image smaller, and the OS runs faster.

The second insight is that the guest OS does not control physical resources like hard drives, network cards, and monitors, and therefore, many drivers and algorithms can be removed from the guest OS. Optimizations of space allocation on hard drives makes little sense, especially since the hypervisor's underlying filesystem may be HDFS. And it is highly unlikely that physical monitors will be allocated to VMs even if they may run components of gaming applications. Moreover, may OSes include legacy drivers and protocol support plugins, which makes sense if their users may run some outdated hardware, however, in a datacenter all hardware specifications can be homogenized, thereby removing the need for legacy support in guest OSes.

The final, third insight is that selecting process VMs and supporting libraries that are installed on top of the guest OS depend on the application's requirements. Installing .Net framework as part of Windows guest OS will simply bloat the VM if it runs a Java application. Some applications may need specialized services that may not necessarily run in the same VM. For example, an application may send its data to some other VM via an RPC that hosts an interface to a relational database, which may be a VAP. Thus, configuring the entire software stack in addition to the guest OS to match the computing needs of the hosted application results in a smaller and faster VMs that are easy to control and manipulate in the datacenter.

[4] `https://www.vmware.com/pdf/vmotion_datasheet.pdf`
[5] `https://www.microsoft.com/en-us/windows/windows-10-specifications`

Next, we will discuss two *unikernel* OSes – specialised, single-address-space VMs constructed by using a *library OS*, where the kernel is implemented as a shared library that can be linked to by user programs that run in the user space [83, 94]. Doing so introduces a great flexibility in customizing the OS, since only those modules that are needed by the programs will be linked to them from the library OS. The resulting image is slimmer and has a better performance.

8.2 OS^v

OS^v is a Linux-based JeOS designed to run in a guest VM on the cloud infrastructure and it supports different hypervisors with a minimal amount of architecture-specific code [83]. OS^v and its VAP images are publically available [6].

A key idea of OS^v is to delegate many functions of the OS kernel to the type-1 hypervisor and run OS^v as a single application in VMs that are hosted by this type-1 hypervisor. The address space is not partitioned in OS^v – all threads including the OS^v kernel run in the same address space and they have a direct access to the system calls that traditionally resulted in traps passing the control to the OS. With the direct access, the overhead is removed of copying system call parameters from the user space to the kernel space. OS^v uses the open-source Z file system manager that can maintain a 128-bit logical volumes that can store up to one billion TB (i.e., a zettabyte) of data. File integrity is ensured by using checksums. Interested readers can download and study the source code of ZFS[7].

Question 3: Does the OS^v design violate Popek-Goldberg properties?

In OS^v, thread scheduling and synchronization mechanisms are implemented using lock-free algorithms based on atomic instructions and cheap and fast thread context switches. Recall the LHPP problem that we discussed in Section 7.6. In OS^v, a pre-empted vCPU cannot lead to wasted CPU cycles, since other vCPUs use lock-free algorithms that will not let those threads to wake up if there is a vCPU that holds the lock. The scheduler creates a run queue for each vCPU that keeps references to runnable threads for each vCPU. Since some vCPUs may have many more threads to run than the others, OS^v runs an LB thread for each vCPU that wakes up approximately every 100ms, checks the queues and picks a thread for a vCPU whose queue is longer and schedules this thread for execution on some physical CPU.

A very interesting aspect of OS^v is that user programs have the direct access to the page table, which is something that is done via privileged instructions in traditional OSes. Recall that the MMU segments the virtual memory in pages whose virtual addresses are mapped to real addresses in the physical memory and the MMU swaps pages on demand. Consider that process VMs, e.g., the JVM have their own MMUs that have internal structures to keep track of memory usages, essentially duplicating some functionality of the OS MMU. By allowing the process VMs to access the OS

[6] `http://osv.io/why-new-os/`
[7] `https://github.com/zfsonlinux`

MMU, memory references can be read and updated without duplicating the bookeeping structures, leading to the improved performance and smaller memory imprint.

OS^v introduced a new channel to allow user programs to obtain information from the OS about memory usage and also (de)allocate it on demand. This channel is implemented in *shrinking* and *ballooning* interfaces of OS^v. Consider the allocation of memory when running a Java program. Using the command-line option to allocate the maximum size of the heap memory often results in either under- or over-utilization. Alternatively, the shrinker interface allows the program to register its callback methods with OS^v, which it will call when certain thresholds are reached (e.g., the amount of the available memory is too low), and the program provides the functionality in the callback to release the unused memory. The reader can view the memory as a fluid matter that flows from one program to the other, and the flow is regulated by OS^v calling registered callback methods.

Question 4: Describe how you would use the shrinking interface in your map/reduce implementation.

To explain ballooning, consider the following rationales. Allocating as much heap memory as possible to a Java application is beneficial, since having a lot of disposable memory reduces the need for invoking a *garbage collector (GC)*, a program that automatically searches unused memory and returns it to the heap as unclaimed by a program. Invoking GC often freezes a process VM and thus negatively affects the performance of the programs that execute on top of this VM. Therefore, it is natural instinct of developers to allocate the maximum available heap memory to their programs.

Unfortunately, allocating too much heap memory to user programs leaves the OS with few options when it needs more memory to accomplish certain tasks like receiving large amounts of data from the network. Without much memory the OS swaps pages in and out of memory, which is an expensive activity and it waste many CPU cycles.

A main idea of ballooning is to allow programs to obtain as much memory as they want initially, since it is difficult to predict how much memory the OS will need. Upon deciding that more memory is needed, the OS will create an object within the process VM and use the reference to this object as available heap memory for its operations. The reader can imagine the process VM as a cloud with the referenced object inside it as a balloon that is inflated or deflated on demand from the OS.

Question 5: Explain what happens to the balloon when the GC kicks in.

To summarize, OS^v does not have any limitation on the language in which a program is written to run it. Creating a VM image with OS^v requires between 10-20MB of the overhead space, and it takes less than ten seconds to build on a reasonably modern workstation. OS^v website contains a number of VAPs with software applications optimized for OS^v. Prebuilt OS^v images can run on Amazon Elastic Cloud (EC2) and Google Computing Engine (GCE) as well as on local computers using Capstan, a tool for building and running applications on OS^v. Moreover, given a small footprint of

OS^v and its impressive performance, it seems as a natural fit for the idea of serverless computing to run short stateless functions that are written in high-level languages like Javascript or Scala. Since OS^v can boot up and shut down very quicky, it is suitable for hosting these serverless functions that take a fraction of the second to execute. Providing *Function-as-a-Service (FaaS)* is important for controlling the cost of cloud deployment, and OS^v is a natural fit for FaaS. Finally, the programming interfaces for administering deployed OS^v images are available, so clients can make remote calls to obtain the statuses of deployed OS^v-based VMs and provide control actions.

8.3 Mirage OS

The approach of Mirage OS is to compile applications with supporting libraries into unikernel appliances thus avoiding a considerable effort to (re)configure VAPs. Mirage unikernel images are compiled to run on top of the Xen, a popular open-source virtualization platform that is commercially supported by Citrix Corporation. Xen hypervisor is type-1, it installed on top of bare metal and it is booted directly using the computer's *basic input/output system (BIOS)* that is non-volatile preinstalled firmware located on the system board and it contains hardware initialization routines that are invoked during the power-on startup and its runtime services are used by OSes and programs. Interestingly, Xen is written is OCaml, a strongly typed functional programming language[8]. The account of building Xen and choosing a non-mainstream language are described by the core engineers who built the original version of Xen and they include the following objectives: performance can be delivered by the code written in OCaml, integration with various Unix programs is facilitated by a simple and efficient foreign-function interface of OCaml, strong type safety of OCaml makes it easier to run the hypervisor that have a very long mean-time-to-failure (MTTF), and strong optimization mechanisms of the OCaml compiler makes it easy to produce compact code [136]. At the time of writing this book, Xen is used by tens of thousands of companies and organizations all over the world.

Mirage OS has a number of interesting features, some of which we discuss briefly in this section[9]. One is zero-copy device I/O that uses a feature of Xen where two VMs may communicate with one another via the hypervisor that grants one VM the right to access memory pages that belong to the other VM. The table that maps pages within VMs to integer offsets is called a *grant table* and it is updated by the hypervisor. Since the notion of user space does not exist in Mirage, applications can obtain direct access to memory pages, thus avoiding copying data from the pages into the user space. For example, when an application writes an HTTP GET request into an I/O page that the network stack writes to the driver, and when a response arrives and is written into a page, the write thread receives a notification to access the page to collect the data. This model is also used for storage, and it enables applications to use custom caching policy, which may make more sense if the semantics of data accesses in the application has patterns that may be exploited for effective caching, rather than delegating it to generic OS policies like *least recently used (LRU)* pages. As we can see, the low-levels of the

[8] `http://ocaml.org`
[9] `https://mirage.io/wiki/overview-of-mirage`

OS and device drivers are not isolated from the application level as we would expect in the OSI seven layer model.

Question 6: What are the key differences between OS^v and Mirage OS?

A number of VAPs have been implemented and evaluated under Mirage. Consider the OpenFlow Controller Appliance, an SDN VAP, where controllers manipulate flow tables (i.e., datapaths) in Ethernet routers. The Mirage library contains modules for OpenFlow parsers, controller, routers, and other networking equipment and a Mirage application can use these libraries to extend the functionalities of these basic virtualized hardware elements. Because the Mirage network stack allows its applications to access low-level components, Mirage VAPs can offer significant extensions of the basic hardware elements with powerful functionalities. Interested readers should check out the Mirage website and follow instructions in the tutorial to create and deploy Mirage unikernel VAPs[10].

8.4 Ubuntu JeOS

We chose Ubuntu JeOS out of dozens of availble JeOSes because of its popularity, well-written documentation, and deep integration with popular virtualization platforms such as VMware[11]. Ubuntu JeOS is an slimmed down distribution of the Ubuntu Server for hosting VAPs and its minimized kernel contains the base elements only needed to run within a virtualized environment. The total size of the installation package of Ubuntu JeOS is around 60Mb, but it may change depending on the type of the VAP that is hosted on top of Ubuntu JeOS.

Ubuntu JeOS is tightly integrated with VMware, which optimized the JeOS for use with its virtualization platform. Creating an Ubuntu JeOS VAP involes a series of simple steps. First, a VM is created for hosting Ubuntu JeOS, with predefined RAM and disk sizes, and the number of vCPUs among other things. Once the VM is defined, a guest Ubuntu Server JeOS image is downloaded from its website and it is imported into the VM. Next, the VM should be assigned a fixed IP address on the network or configured to work with an LB, create partitions on the virtual disk to store programs and data as required by the VAP, and to create a script that prompts the user to create a login name and password. The next step is to install VMware tools and required packages using the command `sudo apt-get`. Next step is to package the application, install it on the VAP, and to configure to use the underlying packages (e.g., create and populate the database that the application uses).

Finally, to conclude the installation, an auto-update `cron` job is created to execute `sudo aptitude installed unattended-upgrades`. Alternatively, updates may be provided manually, especially if the VAP owner wants to ensure that these updates do not remove or add dependencies among packages that can break the

[10] `https://mirage.io/wiki/hello-world`
[11] `https://help.ubuntu.com/community/JeOS`

VAP. With the manual updates, the VAP owner will place `tar` files containing approved updates on a designated server and the `cron` job will access this file at predefined periods of time. The last step is to test these steps, reset the VM for the first user login, and put the VAP on the Internat for download.

8.5 Open Virtualization Format

The process of creating VAPs and JeOSes is conceptually simple: the minimum set of the packages and libraries is selected to run an application or a software stack, bundled together in a VM and released to clients or put into a repository that hosts VM images. However, repeating this process manually for different VAPs is tedious and laborious. To automate it, a standard was created in 2008 that defines how to package and distribute VAPs and software packages in VMs in general, so that tools can be created. The standard is called *Open Virtualization Format (OVF)*[12] and it is maintained by the Distributed Management Task Force (DMTF), nn industry standards organization that studies and improves network-based technologies in open and collaborative review and discussion environment. In case of OVF, commercial companies VMware, Dell, HP, IBM, Microsoft and open source, XenSource, submitted a proposal in 2007 that eventually became the standard for creating and sharing VAPs over the Internet. Currently, major virtualization platform providers released a variety of OVF tools that simplify the VAP construction and distribution.

As defined in the standard, an OVF package consists of the following files: one OVF descriptor with extension `.ovf`, zero or one OVF manifest with extension `.mf` that contain hash codes for individual files, zero or one OVF certificate with extension `.cert` that are used to sign the manifest file and the entire package for security reasons, zero or more disk image files, and zero or more additional resource files, such as ISO images, which are simply binary files that contain copies of sectors from some disc. With OVF, VAPs can be packaged and distributed with the following characteristic features [161].

- VAPs can be distributed over the Internet;
- the VAP installation process is streamlined;
- single and multi VM configurations are supported, and
- independence is guaranteed from host and virtualization platform, or guest OSes.

> **Question 7**: Create a VAP for a firewall and explain how you would deploy it.

An example of the OVF descriptor is shown in Figure 8.1 that is a modified version of the example descriptor whose example is available in VMware documentation[13]. The OVF descriptor is written in the XML format with the tag `Envelope` as the root

[12] `http://www.dmtf.org/standards/ovf`
[13] `https://www.vmware.com/pdf/ovf_whitepaper_specification.pdf`

element of the descriptor in line 1 with the reference to the OVF schema specified in line 2. References to external files and resources are specified with the top tag `References` and they are given between lines 3–5, where only one virtual disk file is specified for brevity. Specific resources are designated with the tag `Section` between lines 6–42, where descriptions of the virtual CPUs, the network, the VAP with the assigned IP address, and the JeOS are given.

Of course, OVF files do not have to be written manually, virtualization platforms provide tools to generate these files automatically. Consider a Java pseudocode example for using VmWare API calls to export OVF of some VM that is shown in Figure 8.2. VmWare is one of the leading providers of virtualization platforms at the time when this book is written, and it offers a powerful SDK to access and manipulate entities in the datacenters that deploy the VmWare platform. To write Java programs using the VmWare Java API calls, one must import the package `com.vmware` as it is done in line 1. The next step is to obtain a reference to an object of the class `ServiceInstance` in line 2, the singleton root object that represents the inventory of the VMWare `vCenter Server` that provices a centralized platform for managing VMware virtual environments unified under the name `vSphere`. The reader can access VMware documentation and study the syntax and semantics of various vSphere API calls in greater detail[14].

The hierarchical organization has the root object of the class `ServiceInstance` that contains zero or more `Folder`s, each of them is a container for holding other `Folder`s or `Datacenter`s, which are container object for hosts, virtual machines, networks, and datastores. Each object `Datacenter` contains zero or more objects `Folder`, which in turn contain zero or more objects `VirtualMachine` or objects `DistributedVirtualSwitch` or `ComputeResource`, which in turn contain objects `HostSystem` and `ResourcePool`, which in turn contains zero or more objects `VirtualApp`, which contain object `VirtualMachine`. vSphere API calls are designed to enable programmers to navigate the inventory hierarchy programmatically. In line 3 we obtain a reference to the object `VM` using the method `obtainInstanceReference` – this method does not exist, we invented it for brevity to circumvent navigating the inventory hierarchy. In line 4 we obtain the object that represents an OVF file and in line 5 we create an object that designates OVF descriptor parameters. We connect these objects in line 6 and then in line 7 the VM is accessed and its configuration is retrieved and put into the OVF descriptor object that is returned in line 9. Samples of code in different languages that use the VMware SDK are publically available[15].

8.6 Summary

In this chapter, we presented the concepts of virtual and software appliances. We showed how useful they are for forming units of deployments in the cloud, where they can be combined in virtual clusters and datacenters. We review Just Enough OSes

[14] `https://www.vmware.com/support/developer/vc-sdk/visdk400pubs/ReferenceGuide/vim.ServiceInstance.html`

[15] `https://code.vmware.com/samples`

```
<ovf:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:ovf="http://schemas.dmtf.org/ovf/1/envelope" ovf:version="0.9">
 <References>
 <File ovf:id="file1" ovf:href="vmdisk1.vmdk" ovf:size="180114671"/>
 </References>
 <Section xsi:type="ovf:DiskSection_Type">
 <Info>All virtual disks</Info>
 <Disk ovf:diskId="vmdisk1" ovf:fileRef="file1" ovf:capacity="10000000"
 ovf:format="http://www.vmware.com/specifications/vmdk.html#sparse"/>
 </Section>
 <Section xsi:type="ovf:NetworkSection_Type">
 <Info>List of logical networks used in the package</Info>
 <Network ovf:name="VM Network">
 <Description>The network for the email service</Description>
 </Network>
 </Section>
 <Content xsi:type="ovf:VirtualSystem_Type" ovf:id="Email Appliance">
 <Info>The Email appliance</Info>
 <Section xsi:type="ovf:ProductSection_Type">
 <Info>This appliance serves as an email server</Info>
 <Product>Email Appliance</Product>
 <Vendor>Lone Star Consulting</Vendor>
 <Version>1.0a</Version>
 <ProductUrl>http://www.url.com</ProductUrl>
 </Section>
 <Property ovf:key="emailvap.ip" ovf:defaultValue="192.168.3.101">
 <Description>The IP address of emai; appliance</Description>
 </Property>
 </Section>
 <Section xsi:type="ovf:VirtualHardwareSection_Type">
 <Info>1000Mb, 2 vCPUs, 1 disk, 1 nic</Info>
 <Item>
 <rasd:Caption>2 vCPUs</rasd:Caption>
 <rasd:Description>Number of vCPUs</rasd:Description>
 <rasd:ResourceType>3</rasd:ResourceType>
 <rasd:VirtualQuantity>1</rasd:VirtualQuantity>
 </Item>
 </Section>
 <Section xsi:type="ovf:OperatingSystemSection_Type">
 <Info>Guest Operating System</Info>
 <Description>Ubuntu Server JeOS</Description>
 </Section>
 </Content>
</ovf:Envelope>
```

Figure 8.1: Example of the OVF description of some VAP.

(JeOSes) and unikernels as mechanisms for creating slimmed down versions of OSes for virtual appliances. After studying OS^{v}, Mirage OS, and Ubuntu Server JeOS, we

```
import com.vmware.*;
ServiceInstance inst = new ServiceInstance(url,uname,pswd,true);
ManagedEntity vm = inst.obtainInstanceReference("VirtualMachineName");
OvfFile[] ovfData = new OvfFile[0];
OvfCreateDescriptorParams ovfDescParams = new OvfCreateDescriptorParams();
ovfDescParams.setOvfFiles(ovfData);
OvfCreateDescriptorResult ovfDesc = inst.getOvfManager().
        createDescriptor(vm, ovfDescParams);
ovfDesc.getOvfDescriptor();
```

Figure 8.2: Java pseudocode for using VmWare API calls to export OVF of some VM.

learned about Open Virtualization Format and how virtual appliances can be distributed with OVF files. We conclude by showing how OVF descriptor can be created automatically from the existing VM using the VMware SDK.

Chapter 9

Web Services

Web services are software components that interact with other software components using document-based messages that are exchanged via Internet-based protocols [49]. The word *service* in the distributed computing context was coined by Gartner analysts W. Roy Schulte and Yefim V. Yatis [108] as a discrete unit of functionality that can be accessed remotely and it is deployed independently from the rest of a distributed application [107]. W3C Working Group defines a service[1] as "an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent." To map these definitions to the RPC domain, a web service can be viewed as an RPC server object which responds to clients' calls made to the web service using the Internet, the global worldwide interconnected computer network that use the general protocols to link computing devices. As such, a cloud datacenter can be viewed as a service whose interfaces expose methods for accessing and manipulating other services (e.g., VMs) using the Internet.

Viewed as a bigger picture, web services are components on the *World Wide Web (WWW)*, a global graph where nodes designate resources or documents and edges specify hyperlinks. Nodes in WWW are usually accessed using the *HyperText Tranfer Protocol (HTTP)*, which is one of the Internet's protocols[2]. Each node can be a client or a server or both in the client/server model. Thus, web services are distributed software components deployed on WWW whose interface methods can be accessed by clients using the HTTP. Method call parameters and return values are embedded in the HTTP messages. Essentially, engineering distributed applications with web services can be viewed as routing text document (i.e., messages) between nodes in WWW where each node is a web service with endpoints described in specifications of the web service interfaces that accept and process the HTTP messages. Thus, creating applications from web services results in a composite web service that orchestrates message exchanges between the composes web services.

[1] https://www.w3.org/2002/ws/arch/

[2] https://www.w3.org/Help/webinternet

9.1 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) combines XML and the Internet transport protocols, e.g., the HTTP to allow computing nodes on the Internet to exchange structured information whose components are assigned semantically meaningful types. The original choice of XML is dictated by its acceptance as one of the universal description formats that can describe any data. Since web servers use HTTP protocol ubiquitously, SOAP is a powerful lightweight protocol that allows clients to access objects anywhere on WWW by sending SOAP-based messages to web servers. Implementing and deploying a platform for web services is straightforward, since only an XML parser, a web server, and SOAP processing libraries are required. Of course, XML can be replaced with some other document-based format like JSON, or XML messages can be compressed before transmission to improve the efficiency of message transfers.

The main goal of this section is to state key principle behind SOAP organization and show its strong connection to the RPC communication mechanism. Interested readers can find in-depth information in the SOAP specification[3] and there are many tutorials and books that cover various SOAP-based technologies with multiple examples of their usage. We recommend that the reader starts with *Java Platform Enterprise Edition (J2EE)* using tutorials located at the corresponding website managed by Oracle Corp.

A dictionary defines a document as "a piece of written, printed, or electronic matter that provides information or evidence or that serves as an official record..." in some language. The language of a web service is defined by its interfaces and the methods that they expose as well as the input parameters and return values that are produced as results of the invocations of these methods. Some sequences of method invocations are valid, others are not as defined by the protocol of interactions with web services. A document-based interaction with a web service is a set of sentences that contain sequences of the identifiers of methods and their interfaces that should be invoked or their return values.

The RPC model is based on the abstraction of a single method call from a client to a server object, i.e., the client's stub constructs a document at the finest level of granularity with a single method call. Opposite to that, a large document can include not only batches of method calls to interfaces of a single web service, but also complex control and data flow instructions that specify how to route document messages in case of failures or in response to certain return values. Consider a mortgage loan approving application where the next function call depends on the value of the FICO credit score that measures the credit risk of a customer. If the FICO score is below some predefined threshold, then one remote procedure is called, which produces a rejection notification to the customer, otherwise, a different remote procedure is called to proceed with the mortgage approval. Sending the return value of the FICO score to the client program and allowing it to make the next RPC results in a redundant round trip of messages that negatively impacts the performance of the loan application.

Instead, a document-based protocol can be used to orchestrate the execution of an application that consists of multiple web services. Moreover, a SOAP-like document can specify batched executions, where the return values of the execution of one method

[3] `https://www.w3.org/TR/soap/`

```
1  POST /DOSOAP/object=WSrv HTTP/1.1 User-Agent: UserName/5.1.2
2  (Linux) Host: http://www.cs.uic.edu
3  Accept: text/*
4  Content-Type: text/xml Content-length: 500 SOAPAction: WSrv#Add
5  <SOAP-ENV:Envelope xmlns:SOAP-ENV=schemas.xmlsoap.org/soap/env
6  SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding>
7          <SOAP-ENV:Body>
8                  <ns:Add xmlns:ns="WSrv">
9                          <x>3</x>
10                         <y>7</y>
11                 </ns:Add>
12         </SOAP-ENV:Body>
13 </SOAP-ENV:Envelope>
```

Figure 9.1: An illustrative example of a SOAP request to call the method Add of the class WSrv that takes values of two integer parameters x and y.

of some web service can be used as the input to some other methods of the same web service. Doing so may significantly improve the performance of applications, since the number of roundtrip request/return messages between clients and the web service can be reduced significantly. Unfortunately, the basic RPC model cannot be easily extended to the document-based model because its core abstraction targets the complexity of a single method call in the distributed environment. Therefore, web services and HTTP-based document protocols while sharing the same client/server model with the RPC, represent a significant conceptual departure from the single distributed method call abstraction.

Question 1: Explain how exception handling is performed in a batched RPC call when an exception is thrown during execution of some function in a batch.

A set of rules for conveying messages between nodes is called a *binding* and the SOAP protocol binding framework defines bindings and how nodes can implement them. Many SOAP binding implementations exist and XML provides a flexible way to encode binding rules and constraints. For example, HTTP binding defines how SOAP calls are embedded into an HTTP message that contains the SOAP envelope that contains the SOAP body element describes the call made over the network.

Let us convert the example of XML-RPC request shown in Figure 6.4 into a simplified SOAP request that is shown in Figure 9.1. We assume that the actual web service is registered with some web server at a known *Universal Resource Locator (URL)* . Recall that a URL is a subset of URI that is an instance of the following specification: `protocol:[//authority]path[?query][#fragment]`, where a `protocol` is one of the allowed communication request/response schemes usually registered with the *Internet Assigned Numbers Authority (IANA)* like `http` or `ftp`; `authority` is an optional component of the specification defined as `[username@ ]hostname[:port]` that may contain the user name, the name of the host or its IP address and the port it is bound to (separated from the rest of the information in

authority with the colon); `path` is a sequence of names separated with the forward slash; `query` is an optional element that defines a sequence of key-value pairs that can be viewed as named parameters to a function call and their values; and finally the element `fragment` that specifies a tagged resource within the resource defined by the `path`. For more information, we refer the reader to as the corresponding *Requests for Comments (RFC)* and other Internet standards.

In the SOAP request shown in Figure 9.1, the web server that host the web service `WSrv` is located at `http://www.cs.uic.edu`. The web service exports the method `Add` that takes two parameters named `x` and `y`. Lines 1-4 contain the HTTP header that specifies that it is a POST request. According to RFC 2616[4], the request POST tells the web server to "accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line." That is, the action of the web server depends on the type of the request. As part of the HTTP header, it is specified that the object of interest is `/DOSOAP/object=WSrv` and it is located on the UIC server. The SOAP action is defined as `WSrvAdd` where the hash sign designates the fragment of the resource `WSrv` that is its method `Add`. Thus, we can see that the resource abstraction where it is described by a URI is concretized in the HTTP request with specific bindings to the SOAP message that is described using the XML notation.

The SOAP message is embedded in the SOAP envelope in lines 5–13. The attribute `xmlns` specifies the URL to the namespace data that define XML elements and attributs for the SOAP message as well as the encoding format. The actual document payload for the RPC call is contained in lines 7–12 where the SOAP body is given. In line 8 the method `Add` is specified and in line 9 and line 10 the values of its parameters `x` and `y` are specified respectively. Once the SOAP message is delivered to the web server, its SOAP runtime identifies the service that processes the incoming requests.

Question 2: Describe how you would build an RPC protocol around SOAP.

To use SOAP, one does not have to create an application as it was done in Figure 6.4, but instead a popular command-line utility named `cURL` for accessing resources and transferring data using a wide variety of protocols and platforms to which `cURL` is ported[5]. The template for this call is the following: "`curl --header "Content-Type:text/xml;charset=UTF-8" --header "SOAPAction: SOMEACTION" --data @FILENAME WSENDPOINT`" without the outer quotes. where `SOMEACTION` is the name of the web service and its method to call, `FILENMAME` is the name of the local XML file that contains the body of the SOAP message, and `WSENDPOINT` is substituted with the URL to the web server that hosts this web service. The utility `cURL` constructs a SOAP document and submits it, waits for a reply and outputs it to the console.

Since SOAP is implemented over stateless transport protocols, it is inherently *stateless*. It means that after a method of a web server is invoked and the results are returned

[4] `https://www.ietf.org/rfc/rfc2616.txt`
[5] `https://curl.haxx.se`

to a SOAP client, the web server object does not keep any information related to this method invocation. Implementing a calculator application using a stateless protocol is a good choice, if the calculator does not have the function memory. However, implementing stateful applications where the information of the previous request/response operations is kept requires using some persistence mechanisms to save the intermediate state of the web server and the previous document exchanges with each of its clients. Doing so imposes a significant overhead on the implementation and the runtime of stateful distributed applications that consist of web services. We will revisit the question of statefulness in the rest of the book.

9.2 Web Services Description Language (WSDL)

With SOAP it is easy to form and send a message to invoke a method of a web service that is not deployed on the target web server. Worse, it is possible to send a message that uses a wrong value type (e.g., a string instead of integer) for a specific method parameter of a web service that is deployed on the target web server. Whereas in the former case the response will will the code 500 indicating the absence of the requested resource, the latter may produce an incorrect result and it will take a long time to debug the distributed application to pinpoint the fault.

> **Question 3**: Is it possible to implement an implicit conversion between different types of the parameter values. For example, if the expected parameter type is `string` and the SOAP request sends an integer value, is it possible to reconcile this discrepancy automatically by converting this integer value into its `string` equivalent? What problems will this implicit conversion lead to?

Recall the *Interface Definition Language (IDL)* that we discussed in Section 4.5. One of its main function is to ensure type correctness between the client and the server procedures. Invoking an IDL compiler on an IDL specification results in generating client and server stubs in a given programming language with the procedure declarations in the generated header files that force client and server programmers to adhere to the same signature of the procedure. If the client programmer mistakenly passes a string variable instead of the integer as a parameter to the procedure call, the compiler will detect this error statically and inform the programmer. However, this is not the case with web services, since they are built and deployed autonomously and independently from client programs.

Web Services Description Language (WSDL) is a W3C specification for an IDL for web services, which are defined as collections of *ports* and messages as abstract definitions of data that are sent to these ports [6]. Port types specify collections of operations that can be performed on these data. Just line in IDL, there are no concrete definitions of how the underlying network transfer of data is performed or how the data of different types are represented during the wire transfer. Specific protocols that governs data

[6] `https://www.w3.org/TR/2001/NOTE-wsdl-20010315`

transfers from and to ports can be defined separately and linked to WSDL documents using its binding mechanism.

Consider an illustrative example of a WSDL document for a web service in Figure 9.2. On the left side of this figure, the web service for configuring a font on a printer is represented as a Java class named `Printer` that has one public method `SetFont`. Its parameter takes the value of the name of the new font and returns the name of the old font. Assuming that this class is deployed as a web service, we want to show how WSDL can describe its structure.

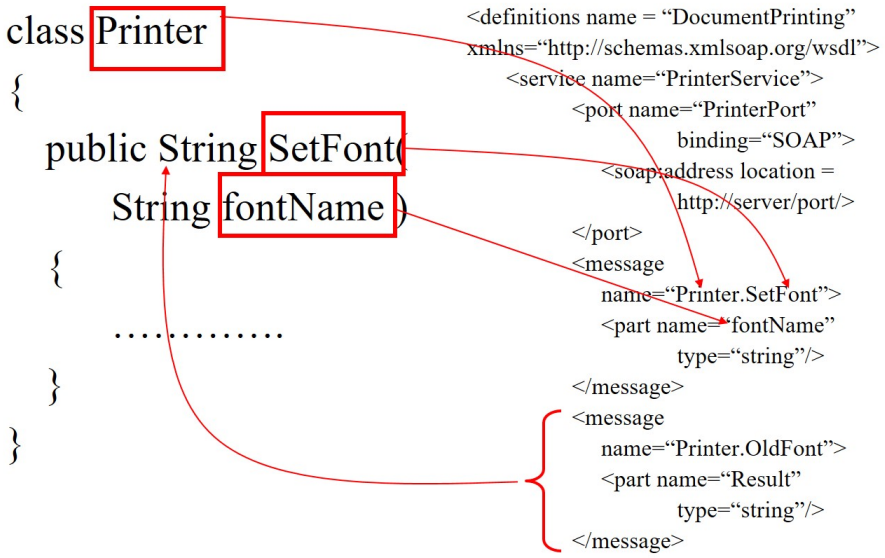

Figure 9.2: A simplified example of mapping between WSDL elements and the source code of the Java class web service.

WSDL document is a text file that contains a set of definitions in the XML format using the names that are defined in the WSDL specification. The document has the root element `definitions` whose attribute `name` is assigned a semantically meaning name of the web service, i.e., `DocumentPrinting`. It also defines the attribute `xmlns` that points to the WSDL schema that defines the namespace. Child element `service` contains the definition of the web service `PrinterService` and there can be one or more web service definitions in a WSDL document. In our case we name this service `PrinterService` and it is mapped to class `Printer`.

A web service is viewed as an endpoint that is assigned to a specific port that receives messages and responds with messages using a binding to some specific protocol. In our case, the web service `PrinterService` has one port that we name `PrinterPort` and it corresponds to the invocation of the method `SetFont`. There are two messages that are associated with this port: the message `Printer.SetFont` whose part is `fontName` of the type `string` and the message `Printer.OldFont` whose part is named `Result` of the type `string`. The port is given the binding to

the SOAP, i.e., the messages are delivered and transmitted encapsulated in the SOAP envelope using the HTTP. To interact with a web service, its clients perform operations, which are abstract descriptions of actions supported by the web service. In our example, the operation is transmitting the SOAP message `Printer.SetFont` to the port `PrinterPort` and receiving the reply message `Printer.OldFont`. As you can see, these descriptions are abstract because they do not contain implementation details of how the web service infrastructure translates these messages into low-level operations on remote objects and their procedures.

Thus, WSDL allows stakeholders to obtain an abstract description of the interfaces of web services. Each web service makes computations in response to input messages and produces output messages, where the collection of input and output messages for a service is called an *operation*. *Port type* is an abstract collection of operations supported by one or more endpoints, e.g., a printing port type that collectively describes a range of services offered by multiple endpoints. A port is a single endpoint defined as a combination of a binding and a network address. Each port is associated with some binding that defines a concrete protocol that specifies how operations are accessed. The reader may visualize a WSDL document as a virtual dashboard on the Internet where web services represent grouped sockets, each of which represents some port, clients plug in virtual wires with connectors into these sockets and predefined messages are transmitted and received via these wires. Whereas the source code of the physical web services may be modified on demand, the WSDL interfaces remain immutable thereby allowing clients to access web services on demand.

> **Question 4**: How to maintain the immutability of WSDL interfaces if some of its methods must change due to new requirements?

Finally, WSDL enables reuse of abstract definitions across multiple applications and underlying platforms. Since WSDL defines a common binding specification, it is easy to attach different protocols to an abstract message or operation, thus interoperating heterogeneous applications and platforms and creating a larger composite distributed applications. Moreover, WSDL is not a new type definition language, it is based on *XML Schema Definition (XSD)* to define pluggable type systems[7]. In general, schemata describe shared vocabularies for definiting the structure, content and semantics of structured documents. It means that messages and operations of web services are defined in terms of existing XML schemata and they can be shared across all stakeholders on the Internet.

9.3 Universal Description, Discovery and Integration

The ease with which web services can be created and deployed on the Internet raises a question about their discovery. Consider a situation where a team is creating a webstore distributed application and a component should be developed that computes shipping

[7] `https://www.w3.org/XML/Schema`

rates for products. Since many webstore applications have already been built, team members have a strong feeling that creating this component from scratch would be a waste of time, since it is highly likely that shipping companies like UPS already created shipping rate web services for use by many companies and organizations all over the world. Therefore, the question is how to find and integrate these web services.

Universal Decription, Discover and Integration (UDDI) is a specification for a web services directory project that was initially viewed as Yellow Pages for electronic commerce applications[8]. The main idea is to create a global public registry where everyone can register components, for example, a company can make its web services accessible to other companies. A few commercial companies and many organizations participated in the development of UDDI specification, and implemented their versions of UDDI as web services whose clients can use to locate other web services that implement specific requirements for software projects.

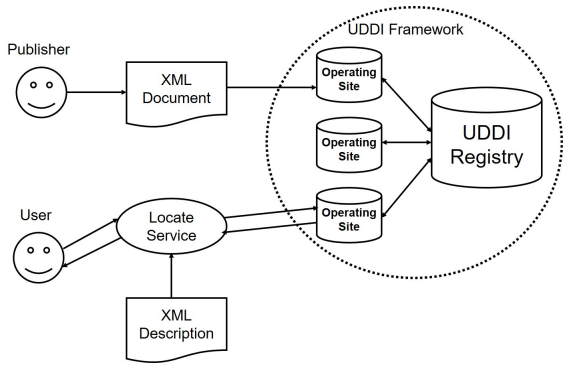

Figure 9.3: The UDDI interaction model.

Question 5: Describe an example of a search query to obtain WSDL of a web service that matches some business requirements.

The UDDI model is shown in Figure 9.3. A publisher creates an XML document (e.g., WSDL) that describes the web services that are made available to users. This document is stored onto one of Operating Sites that are linked to the central UDDI Registry, a key/value store that is updated with information coming from all operating sites. When a client/user make a call to locate a web service using some UDDI API function, the information about the properties of the desired web service is passed as a parameter to this function. Depending on the sophistication of the search algorithm, WSDL documents describing web services that are matched to the query are returned to the client who can make choices about what web service to use in her applications.

UDDI specification defines four types that describe any web service: ① business information type is the White Pages of UDDI that describe what business offers web services, its organizational structure, and other information like unique identifiers; ② service information type groups web services by the business process they belong to and some technical characteristics; ③ binding information that allows a user to locate

[8] `http://www.uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm`

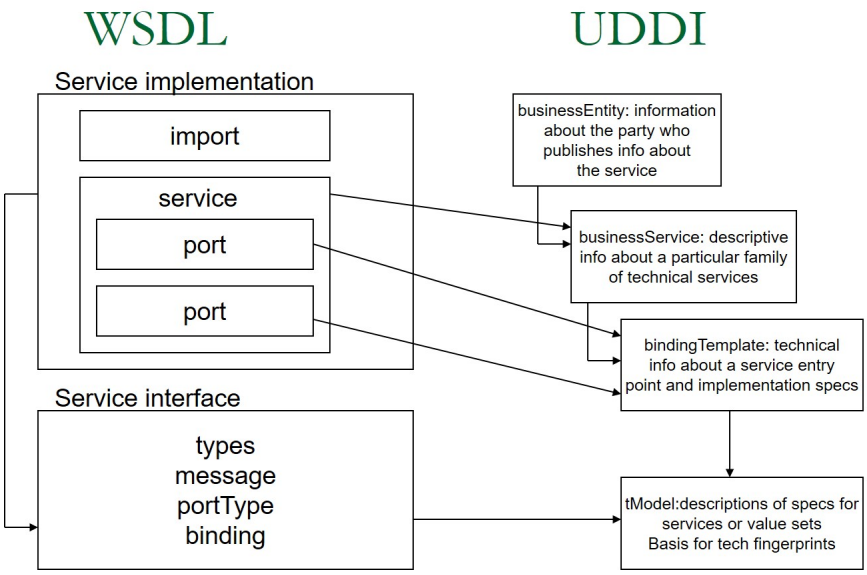

Figure 9.4: Relationship between WSDL and UDDI.

and invoke methods of exposed interfaces of remote web services, e.g., the URI of a web service and platforms that it runs on; and finally, ④ service information describes various aspects of a web service, for example, compatibility with a specification of some other web service the former depends on. Each `<bindingTemplate/>` element of the types contains a list of references to related specifications that can be thought of as a technical fingerprint of a web service or specification metadata. This information is contain in tModels to help developers to obtain a common point of reference that allows compatible web services to be discovered and used. Of course, it requires web service vendors to supply sufficient information about the specifications in a format that is known, can be parsed using some standardized tools (e.g., XML parsers, and queried.

Question 6: Explain the relation between UDDI and location and migration transparencies.

Relationship between WSDL and UDDI documents is shown in Figure 9.4. WSDL document information about the corresponding web service that it describes is reflected in the business service type of the corresponding UDDI; the information about ports is placed in the binding template; and the rest of the information about messages and port types is placed in the tModel sections. Thus, WSDL with UDDI represent a powerful mechanism for sharing and reusing web services on the Internet scale.

9.4 Representational State Transfer (REST)

Despite their benefits, SOAP documents are large compared to RPC even if one accounts for the possibility of compressing them when transmitting between clients and web services. Bloated messages increase latency communication time, impose processing overhead, and are difficult to analyze especially if they are routed across multiple endpoints each of which absorbs some SOAP messages and produces new ones that may contain the information from the absorbed SOAP messages plus some additional data computed by the invoked methods of the web service.

In the stateless client/server communication model, clients send requests to remote servers that invokes procedures on the behalf of clients and returns the results of these procedure invocations to the clients. When a user directs her browser to some URL, she accesses an HTML page. The user is a client and the remote web server invokes internal procedure calls to deliver the HTML content to the user's browser. One can think of the HTML document encapsulated in a web service that provides it by exposing a method to clients in its interface. Or one can think of a servlet, a in-web-server resident software component that extends the capabilities of the web server, which responds to the request from a client and produces and HTML string that it returns to the client. The reader can see that despite different implementations these examples share a common abstraction where a target remote object is a generalized *resource*. All references of clients to target remote objects are viewed as accessing and manipulating resources.

Question 7: Write an IDL or WSDL for a generalized resource.

The resource abstraction is the essence of the *Representational State Transfer (REST)* architectural style for building distributed applications [50]. In functional terms, a resource is a function that maps a notion of a resource to the set of values that include the resource identifier (i.e., its URL or URN) and a specific representation (e.g., HTML document or WSDL). An unrealized resource is a mapping to the empty set. The mapping can change over time, where the same resource may be represented by an HTML page first and a web service later. In general, the representation of a resource contains the content of the resource and its metadata. Since REST is stateless, the states of clients and resources cannot be maintained by some third parties like resources maintaining information about the history of the requests received from different clients. Some part of the state may be cached, i.e., the data can be stored in a special location with the intention that the next time client needs this data it can be delivered to it faster compared to making a separate request to a remote object. Summarily, designing applications using REST architectural style enables stakeholders to define clearly what resources are and how to access and manipulate them in a simple and straightforward way using messages in the client/server context.

Since resources can be viewed as abstract nodes on WWW, accessing and manipulating them can be done using REST with standardized HTTP commands as defined in the HTTP specification in RFC 2616[9].

[9] `https://www.ietf.org/rfc/rfc2616.txt`

GET command retrieves information from a resource that is identified with some URI that has the same specification that we specified above following specification: `protocol:[//authority]path[?query][#fragment]`. For example, REST command `https://owner-api.teslamotors.com/api/1/vehicles` shows all Tesla vehicles that the client owns, assuming that the authorization is provided in the part `authority` of the URI. Conditional `GET` using the data put in the axilliary fields regarding the date of the last modification, matching to some values or a range of values to return the state of the resource only when these data are evaluated to true. For example, the Tesla `GET` request can be made conditional by requesting information only if it was modified since yesterday and for the model S only.

HEAD command results in the return value that contains only the header. For example, the return value `200 OK` signals that the web service is available and can respond to requests. It may be used to check if the authorization is still valid, or if there are changes to the representation of a resource, among other things.

POST command contains the information about some entity that is related to the existing resource. It may contain some metadata to annotate the existing resource; it may contain the representation of a new resource (e.g., a new HTML file or a message on a social media site); or it may contain the information about method invocation of some web service and its parameters values. Abstractly, this command inserts a new resource or updates an existing resource. Executing this command more than once on the same resource may result in a different state of the resource. For example, posting the same message N times on a social media website will result in the appearance of N same messages.

PUT command has the same semantics as the command `POST` except it is *idempotent*, i.e., executing it on the same resource N times will result in the same state of the resource as if this command was executed only once. In the example above, `PUT`ting the same message N times on a social media website will result in the appearance of only one message.

DELETE command deletes a specific resource. Once deleted, the repeated execution of the command on the same resource identifier has no effect.

OPTIONS command requests information about the specified resource. For example, it may return a document analogous to WSDL that specifies interfaces and their methods of the designated web service.

Using the HTTP protocol as the main transport mechanism for RESTful web services adds the benefit of caching. The HTTP specification defines fields in the header of an HTTP request that control how the HTTP responses are cached. For example, the field `Age` specifies the duration in seconds since the information was obtained and the field `Date` specifies when the representation state response was created. Caching may provide stale data, however, it may significantly improve the availability of the system. We will explore the tradeoff between these important properties later in the book.

RESTful web services are easier to make available to clients without creating UDDI registries. All it is required is to report the name of a web service, describe its semantics, provide the URI, and give some examples of input parameters and the corresponding responses. Many commercial companies and organizations release development kits and specifications for their web services that other companies can use. For example, United Parcel Service (UPS), an American multinational package delivery and supply chain management company with the revenue of over USD $65 Billion as of 2017 offers its UPS developer kit for its web services with which clients can embed UPS tracking and shipping information in their applications[10]. The list of available web services include but not limited to address validation, pickup, shipping, signature tracking and many other delivery-specific APIs supported by the UPS web services.

9.5 Integrated Applications as Web Service Workflows

Enterprise Application Integration (EAI) is an inglorious and boring process of linking enterprise-level applications (i.e., large-scale and often distributed applications whose execution is critical to achieve main business objectives in the enterprise) within a single organization together in order to streamline and automate business processes without making major changes to the source code of the integrated applications. Gartner defined EAI as the "unrestricted sharing of data and business processes among any connected application or data sources in the enterprise" [56]. An important element of EAI is to change the functionality of an integrated application by adding new features or changing the data flow among the existing integrated components (e.g., web services) without changing the source code of these components.

> **Question 8**: How difficult is it to change the source code of a large-scale enterprise-level application? What steps are involved in this process?

Consider as an example *E-Procurement Systems (EPSes)*, which are critical application in every business and nonprofit/government organization, since they are used by different stakeholders to acquire products for conducting business. Elaborate EPSes are employed which often consist of different applications assisting different steps of the purchasing process. In EPSes, the *rule of separation of duty* requires that operations be separated into different steps that must be done by independent persons (agents) in order to maintain integrity, or simply put to make it difficult to steal goods and money. With the separation of duty rule in place, no person can cause a problem that will go unnoticed, since a person who creates or certifies a transaction may not execute it.

Consider a typical EPS scenario where employees order items using an electronic shopping cart service of the web-based application *OrgDepot (OD)*. Department managers review selected items in the shopping cart, approve and order them, and enter the ordered items into *Quick Expensing (QE)* to keep track of purchases. The OD service

[10] `https://www.ups.com/us/en/services/technology-integration/online-tools-rates-svc.page`

sends a notification to a company accountant, who uses *Purchase Invoices and Estimates (PIE)* to create invoices for ordered goods. When the ordered goods are received from OD, a receiving agent compares them with the entries in QE. The accountant can view but cannot modify records in QE, and likewise, no other agent but the accountant can insert and modify data in PIE. If the received goods correspond to the records in QE, the receiving agent marks the entries for the received goods in QE and notifies the accountant. After comparing the invoices in PIE with the marked entries in QE and determining that they match, the accountant authorizes payments.

Each of these services can be used as a standalone application, however, it would require a significant manual effort to enter the data into each application and to verify the entries by matching them against one another. Real EPSes are much more sophisticated, since there are many government and industry regulations that govern procurement processes, and these regulations are updated and modified. Correspondingly, EPSes must keep up with these updates and modifications. Unfortunately, modifying the source code of these services is laborious, error-prone, and it takes time, which is critical in the competitive environment where other companies can adjust their services quicker and at a lower cost.

This example shows that it is important to reflect changing business relationships in integrated applications where the distributed state is maintained by two or more services of different business parties and business transactions represent a consistent change in the state of the business relationships. A key to creating flexible integrated applications is to separate the concerns for business relationships among services from the concerns that describe the business logic or units of functionality (i.e., features) in each service. This separation of concerns is easy to achieve in graph representations called *workflows*, where nodes represent operations performed by individuals, groups, or organizations, or machines and edges represent the business relationships between operations. A workflow can be viewed as an abstraction of some unit of work separated in finer granular operational units (e.g., tasks, workshares or worksplits) with some order imposed on the sequence of these operational units.

Question 9: Create a workflow for an EPS and explain how to map its nodes to web services that implement the EPS' functionality.

The structure of many workflow systems is based on the *Workflow Reference Model* that is developed by the *Workflow Management Coalition (WfMC)*[11]. Workflow applications are modeled as collections of tasks, where a task is a unit of activity that is modeled with inputs and outputs. The workflow structure is defined by interdependencies between constituent tasks.

A business process is easy to represent using workflows with starting and completion conditions, constituent activities and rules for navigating between them, tasks and references to applications which should be run as part of these taks and any workflow relevant data that is ingested and manipulated by these applications. Once a workflow is constructed, *the workflow enactment software* interprets it and controls the instantiation

[11] `http://www.wfmc.org`

of processes and sequencing of activities, adding work items to the user work lists and invoking applications, manages the internal states associated with the various process and activity instances under execution. In addition, the workflow enactment software includes checkpointing and recovery/restart information used by the workflow engines to coordinate and recover from failure conditions. Essentially, constructing distributed applications using web services is a workflow wiring activity where the engineer defines a workflow to implement some business applications, then selects web services with specific semantics that satisfy some business feature requirements, and finally, wires these web services using the workflow schema into the application that will run on top of some workflow enactment engine whose job is to deliver the input data to these web services in some order, take their outputs, and continue in this manner until some conditions or objectives are met. Thus, the key is how to express workflows, so that they can be executed by the underlying enactment engine.

9.6 Business Process Execution Language

Business Process Execution Language for Web Services (BPEL4WS) is a language and a runtime for creating and executing distributed applications by specifying business relationships among web services and executing them while coordinating actions and maintaning states. A BPEL program can be viewed as a composite web service that coordinates a flow of data between some other web services. In-depth discussion of BPEL is beyond the scope of this book. The full specification for BPEL4WS is given in the OASIS standard[12].

Consider the most likely simplest example of a BPEL program that implements the RPC-style communication between a client and a server web services as it is shown in Figure9.5. The first step is to to view RPC not as a server function invocation by the client, but as a workflow with two nodes that are designated as BPEL processes and the flow between them that is expressed in terms of input and output messages. In the standard RPC, there is a location in the client program where the server function is invoked and it is referenced by its full name. That is, the client and the server programs are *tightly coupled* via the function invocation and the only way to uncouple these components and change the function invocation to some other function is by editing and recompiling the source code and redeploying the component. Opposite to that, in *loosely coupled* deployment the client and the server programs do not have any information about each other in their source code bases. An external component like a BPEL process will decide how to invoke functions of the server.

Thus, the simplified BPEL program that is shown in Figure 9.5 describes how a client service makes a call to the remote function `someFunc`. In lines 1–3 the BPEL program defines the name of the process and its namespaces. The tag `process` is the root element of the BPEL program that specifies the scope of its internal definitions. The section defined by the tag `partnerLinks` in likes 4–6 specifies the services with which a business relationship is established in the workflow defined by the BPEL program, where roles specify port types. One can think of partnerLinks as endpoints

[12] `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`

```
1  <process name="BPELClientServer" targetNamespace="cloud:RPCExample"
2  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
3  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
4  <partnerLinks>
5    <partnerLink name="clnt" partnerLinkType="Client" myRole="RPClnt"/>
6  </partnerLinks>
7  <variables>
8    <variable name="request" messageType="tns:requestMessage" />
9    <variable name="response" messageType="tns:responseMessage" />
10 </variables>
11 <sequence name="ClientServerSequence">
12   <receive name="RpcCall" operation="someFunc" partnerLink="clnt"
13     portType="tns:Client" variable="request" createInstance="yes"/>
14   <reply name="ReplyRPC" operation="someFunc" partnerLink="client"
15     portType="tns:Client" variable="response"/>
16 </sequence>
17 </process>
```

Figure 9.5: A BPEL program that defines a synchronous RPC between a client and a server.

at which remote objects are deployed and bindings specify how communications are performed using concrete protocols.

The next two sections are defined by elements `variables` in lines 7–10 and `sequence` in lines 11–16. Variables are named locations that store values and in our example we use two variables named `request` and `response` whose specific message types are defined in some WSDLs. The variable `request` is somehow set a value that will be received by the remote function `someFunc` defined in the section `sequence` that is a set of sequential activities `receive` and `reply`. The former activity specifies in lines 12–13 that it receives a message of the type `tns:requestMessage` from the partnerlink `clnt` and it will automatically put the value of this message into the variable `request`. Finally, it will return the reply contained in the variable `response` to the client caller.

Question 10: Describe how you would implement an EPS with BPEL.

A BPEL program can be viewed as a director who orchestrates the performance of constituent web services. A main consideration for using BPEL programs is a degree of centralization in the distributed application. At one extreme, the BPEL program orchestrates the execution of the application by directing messages to web services and invoking their methods. At the other extreme, web services “decide” which other web services they interact with based on the results of computations within web services. For example, a web service that retrieves the FICO score of a customer decides what other web service to send a SOAP message depending on the value of the score. That is, the distributed application is choreographed where each web service has the knowledge of other web services and under what conditions it will communicate with them. Whereas orchestrated applications are easier to reason about, they have the single point of failure – the orchestration BPEL program.

It is important that the client program does not have any knowledge about the server program. The BPEL program implements the workflow based on the notions of ports and data types – it provides location and migration transparencies to allow web services to issue requests and receive reply messages independently of how other web services are created and deployed. The underlying protocols govern how web services are accessed, however, each web service is opaque – its internal implementation is hidden from its users and the users know only of its interfaces and their exposed methods as declared in WSDLs. Building applications becomes akin to wiring components on a dashboard excepts these components are web services, which can be located on servers connected to the Internet around the world.

9.7 Summary

In this chapter, we describe web services, RPC server objects which respond to clients' calls made to the web service using the Internet, their underlying protocols, and how to build applications using web services. Our starting point was the Simple Object Access Protocol (SOAP), a rather heavyweight way to enable web services to interoperate by describing method invocations and data exchanges in XML messages embedded in HTTP. Next, we presented Web Services Description Language (WSDL) as an IDL of web services. A client program can obtain the WSDL for a web service by sending a SOAP request and analyze the interfaces and methods (i.e., messages and ports) exposed by the web service. Aggregating many WSDLs for different web services makes it possible to create a universal database as a web service in Universal Decription, Discover and Integration (UDDI) mechanism, where web services can search for other web services that have desired properties expressed in search queries. As a lightweight alternative to SOAP, we describe the Representational State Transfer (REST) architectural style for constructing HTTP requests to access and manipulate web services. We conclude by explaining how distributed applications can be built as workflows connecting web services.

Chapter 10

Microservices and Containers

What is the right granularity for a web service? How to measure the granularity of a web service? *Lines of code (LOC)* is a very popular measure in systems and software, however, as many generic measurements it hides some important semantics, e.g., how many functional units are implemented in a web service. At one extreme, the entire application can be created as a single monolithic web service. Opposite to it, the design of the application can be decomposed into highly granular web services, where each service may implement only few low-level instructions (e.g., read a line from a file or write a value to a memory location). While a single monolithic web service may be easier to debug, its scalability and efficiency may suffer as well as performance, since it will likely be a bottleneck for processing multiple requests simultaneously. In addition, different features of this web service may be implemented and supported by different geographical entities, thus making it difficult to deploy as a single monolithic unit.

Creating highly granular web services has its disadvantages, most notably tracking deployment and enabling choreography/orchestration of thousands of these services in a single application, performing transactions across the multitude of web services, analyzing logs to debug production failures, and update web services with patches and new versions, among many things. Equally important is the issue of how to deploy highly granular web services – placing each service in a separate VM is likely to result in thousands of running VMs for a single application. Each VM includes its own virtualization layer and the OS thus consuming significant resources, many more that are needed to run a web service.

In the world of transportation, containerization is a freight mechanism where transported goods are packed in *International Organization for Standardization (ISO)*-defined shipping containers that have standardized dimensions and support for keeping the transported goods in the original condition. Goods must be isolated from one another inside a container to prevent their accidental damage during shipping and inadvertent tampering as well as isolated from the external conditions (e.g., rain or high temperatures). Using the transport containerization analogy, a container for service components includes an isolation mechanism that allows services to run independently from one another and the storage mechanism that keeps service image inside the container. VMs are containers, of course, but given how diverse web services are, different types

of containers are needed to host them, and this is what we discuss in this chapter.

10.1 Microservices

Creating monolithic applications, which contain programming modules that are often run in the same address space with explicit hardcoded dependencies, is popular especially during the prototyping phase of the software development lifecycle. The idea is to demonstrate the feasibility of a certain business concept and to receive the feedback from stakeholders. With monolitic application design, programmers are not required to worry about issues that plague distributed computing, e.g., availability, scalability, or performance of the prototypes. However, monolitic applications are difficult to deploy and maintain in the cloud environment for the same reasons that make creating monolitic applications popular – the hardcoded dependencies in the source code prevent stakeholders from quickly adding and removing features and from provisioning resources in a granular manner to the parts of the applications that require different support from the underlying hardware.

Decomposing an application into smaller components or services and deploying them in distributed VMs allows stakeholders to distribute the load effectively among these VMs and to elastically scale up and out these VMs with the increasing workload. Consider a web store where customers search and view products much more frequently than they purchase them. Splitting the web store application into searching, purchasing, and shipping services allows the application owner to scale out searching services to provide fast responses to customers whereas fewer resources are needed for purchasing and shipping services, thus reducing the overall deployment cost while maximizing the performance of the application. Doing so would be difficult if the application was a single monolithic service itself.

Question 1: Explain how to use BPEL to control the granularity of services.

We define a microservice as an atomic unit of functionality that is often implemented as referentially transparent, side-effect-free functions where the outputs of each of the functions depends on all of its inputs and these functions are exposed to clients and a microservice is deployed autonomously and independently from other microservices. As an atomic unit of functionality, a microservice cannot be further decomposed into more granular microservices. Alternatively, a composite microservice is composed of atomic and other composite microservices. When invoking functions of the referentially transparent microservices, the invocations of these functions can be replaced with the values that these functions compute for given inputs. That is, microservices do not modify the global state of the application by writing into memory or some storage except by returning values to clients as messages. Finally, each microservice does not have dependencies on other microservices in an application that require the presence of some third-party microservices for installation and deployment of the other ones.

The state of a microservice is the set of the locations of the memory that it controls and manipulates and the values stored in these locations. Only the instructions of the

source code of the microservice can change its state; other microservices can obtain projections of its state by invoking its exposed functions and receiving the return values. In general, there should be no global memory that can be accessed and manipulated by microservices, since each microservice would be able to affect the execution of other microservices by changing values in the global memory locations. Doing so results in losing the autonomy and increasing the coupling among microservices, since the behavior of each microservice will depend on specific operations performed by the other microservices that change the global state.

Microservices can be coupled indirectly via exposed memory regions when these microservices are deployed on the same hardware platforms. Consider a situation when one microservice writes data in some memory location it owns before the underlying OS switches the execution to some other microservice that runs on the same hardware. If the memory is not erased when switching contexts, which may happen to reduce the context switching time, then the other microservice may read the data since it will own the memory locations during its execution. Therefore, it is important that the underlying platform provides strong isolation of contexts for microservices that will prevent accidental data sharing and data corruption between microservices.

Question 2: How would you create a distributed application where all microservices are coupled dynamically, at runtime.

One of the most difficult to quantify measures for microservices is the measure of granularity. At one extreme, a microservice may expose a single function that returns some constant value. At another extreme, a microservice can export many functions each of which implements a whole business process and all these functions are used in one another. The former results in a large number of microservices that are difficult to remember and programmers are overloaded with information about them. Moreover, latency becomes a primary concern when an application is composed of too many microservices, since the number and the frequency of messages between them results in a significant performance overhead. A rule of thumb is that a microservice implements some basic unit of functionality that may be used in different subsystems and they can be composed to implement larger units of functionality.

Naturally, microservices are frequently implemented as RESTful web services or as software components whose interfaces are exposed via web services. Moreover, a microservice can be implemented as a function, which is invoked in response to some events, e.g., `(ev1,ev2, store)=>store(ev1.time-ev2.time)`. In this case, the function is instantiated at runtime as a lambda function literal that is invoked in response to events `ev1` and `ev2` and another function called `store` is passed as a parameter. The body of the function invokes the function `store` on the input parameter that is computed as the difference between the attribute `time` of these events. Recall that the FaaS cloud computing model is designed to invoke "serverless" functions in response to certain events. Thus, an application may consist of microservices of different granularities that are designed to be hosted in different runtimes and whose execution times can last from microseconds to many days at a time.

Given the diversity of microservices in terms of their execution time and com-

plexity, they require deployment containers that satisfy different properties of these microservices. VMs are heavyweight – they have much longer startup and shutdown times, they occupy much larger memory footprint, and they contain many virtualization layers (e.g., hardware, drivers, OSes) with many utilities and tools that are not needed for most microservices. Clearly, deploying a stateless lambda function that takes only one millisecond to run in a dedicated VM that hosts a guest OS is an overkill – it costs a lot and the performance overhead is significant, especially if this function is called once every couple of minutes or so. Therefore, a variety of containers are created for hosting different types of microservices. We will discuss them in the rest of this chapter.

Question 3: What problems would you have to deal with when deploying an application that is created from microservices that create side-effects?

10.2 Web Server Containers

One of the straightforward solutions is to implement microservices as RESTful services that serve their clients from VMs in which they are deployed. We can pack multiple RESTful microservices in a single VM and expose them via endpoints served from the same IP address and the port. In this section, we will use Twitter Finagle-based frameworks to discuss how to create and deploy RESTful microservices [47].

Consider an example of the implementation of a RESTful microservice with the Scala-based pseudocode in Figure 10.1. This microservice uses a framework called `Finch` that is run on top of `Finagle` and it was released by Twitter Corp[1][2]. The core modules from `Finch` and `Finagle` are imported in line 1 – line 3. The main singleton object `FinchMicroservice` is declared in line 4 and it contains the code for defining endpoints and the functionality for the microservice that handles student information. Each endpoint is defined by the routing path that clients specify to reach the desired service. In line 5 the variable `enrollStudent` specifies that in response to the request `Post` that is applied to the path `/student` at some URL, say `https:\\www.uic.edu:8080\students` that designates the given web service, the code to enroll the student will be executed in line 6 and the newly generated *University ID (UID)* is returned as part of the HTTP message to the enrolling student in line 7 by invoking the function `Ok` with the parameter `uid`.

The web server is instantiated in like 14 by invoking the method `serve` on the object `Httpx` whose first parameter includes the port number on which the web server is listening. That is, it is possible to start multiple web servers within the same VM, and each web server will listen to HTTP requests on different distinct ports. The address of the web servers is given by the immutable part of the URL `https:\\www.uic.edu:<portnumber>` and for each web server the mutable part is where the corresponding `portnumber` is placed. The endpoint of the web service is defined with a *route*, which is specified in `Finch` with the combination of path names, HTTP

[1] `https://twitter.github.io/finagle`
[2] `https://github.com/finagle/finch`

```
1  import com.twitter.finagle.Httpx
2  import io.finch.request._
3  import io.finch.route._
4  object FinchMicroservice extends App {
5    val enrollStudent = Post / "students" / string /> ( SSN=>
6          /*code to enroll a student*/
7          Ok(uid.toString))
8    val studentInfo =
9       Get / "students" /> /*return the list of students*/ |
10      Get / "students" / long /> ( uid => /*return student's info*/ ) |
11      Put / "students" / long /> ( uid => /*update student's info*/ ) |
12      Delete / "students" / long /> ( uid => /*delete the student*/ ) |
13      Post / "students" / long / "awards" /> (uid => /*award winning news*/)
14   val ws = Httpx.serve(":8080",(enrollStudent :+: studentInfo).toService )
15   /*wait for the exit signal*/
16   ws.close() }
```

Figure 10.1: Scala pseudocode example of a RESTful web service using Twitter Finch.

methods, and `Finch` combinators. The route specification is used in *matchers* that are methods for finding the endpoint using the route specification. For example, in line 5 the route is given by the method `Post` followed by the combinator `/` called *and then*, that is followed by the string variable `students` that is in turn followed by *and then* combinator and the *social security number (SSN)* of the student and finished with the map combinator `/>` that takes the input values for the `SSN` from the route specification and passes it as an input parameter `SSN` of the programmer-defined function in line 6 – line 7 that creates a record for the student. Thus, the variable `enrollStudent` specifies a route to creating a student's record for this microservice.

> **Question 4**: Describe different ways of combining microservices into composite services using Twitter Finch.

The other route, `studentInfo` is shown in line 8 – line 13 and the reason for creating a separate route is because it deals with the records of already created students opposite to the route `enrollStudent` that deals with requests of students whose records haven't been created yet. The route contains HTTP methods `Get`, `Put`, `Delete`, and `Post` for obtaining the list of students in line 9, obtaining the information on a specific student identified by her `uid` in line 10, updating the information about a specific student in line 11, deleting a student's record in line 12, and creating a subsection in the student's record to list all her awards. The keywords `long` and `string` are extractors whose goal is to convert a sequence of bits in the URL designated by ad then separators into a value of the designated type. Tthe combinator `=>` is the function mapping operator that separates the input and the result of the programmer supplied functions for the microservices. Finally, the combinator `:+:` combines routes in the second parameter to the method `serve` in line 14 thereby creating a web server object that listens to clients HTTP requests and executes functions in response to them.

The web server waits in line 15 for a shutdown signal and once received, it closes the connection in line 16 after it cleans up necessary resources.

This example illustrates three important points. First, a web server is a distributed object that contains (micro)services that are defined as combinations of routes to endpoints and user-defined functions that are invoked in response to clients's HTTP requests. Web servers with contained microservices can be deployed in VMs with all necessary dependencies and load balancers can distribute requests among replicated web servers in multiple VMs thus scaling out applications. Second, depending on the functions performed by microservices, they may be deployed together locally in web servers to minimize communication time among them. As we already know, localities impact the latencies significantly.

> **Question 5**: Discuss a construction of an optimizer that would re-arrange the assignments of microservices to physical servers in the cloud to reduce the overall latencies and decrease the network traffic.

Finally, adding new routes is easy, since the old ones can be kept intact with functions redirecting requests from old routes to new ones if necessary. Unfortunately, it may not be possible to deploy many microservices in different web servers in the same VM for many reasons, most important of which is the lack of isolation among microservices and the difficulty to scale up VMs, since some microservices may not a lot more resources to perform their functions than their counterparts. In addition, web servers will stay active in the VM all the time consuming resources, even if requests from clients arrive infrequently. Equally important is the support for interacting with persistent storages (i.e., files and databases). Web servers are agnostic as containers to what database support is required for microservices, however, some other containers provide this level of support.

10.3 Containers From Scratch

With all benefits of program isolation and ubiquity of deployment options, VMs introduce an unavoidable overhead due to the presence of the hypervisor and the emulation layers to enable programs to run on different hardware. Removing this overhead while preserving the benefits of VMs enables microservices to deploy and execute faster and with proper isolation from one another.

To remove the hypervisor overhead, microservices should be run directly on top of the operating system which is installed and run directly on top of bare hardware rather than a hypervisor. However, this idea reduces the virtual computing environment to a standard OS multitasking environment where multiple processes share the same hardware and software packages. Of course, the OS kernel switches among these processes giving each process some time slice of the CPU and these processes can access various resources that they share among themselves, sometimes unintentionally. Besides protected memory spaces, little isolation exists when it comes to shared files in the filesystem (e.g., libraries and system files). Using access mechanisms to enable some

processes share data while preventing accesses for other processes is complicated in a dynamic environment where it is unknown in advance what microservices will be installed on each physical computer. Moreover, it is difficult to achieve location and migration transparencies, since when moving a microservice to a different computer it is important to ensure that all needed libraries and configuration files are installed there, otherwise the microservice will not run. Finally, it is important to regulate how many resources each microservice can consume, since the owners of microservices pay for the consumed resources and allowing a cheaper service to utilize the same CPU equally with a more expensive microservice whose owner paid for scaling up.

Question 6: How can splitting a microservice into more basic microservices that are organized to perform the function of the original microservice, increase or descrease the overall performance of the microservice?

The idea of containerization of services is twofold: ① to provide a virtualized environment for each microservice with a controlled portion of resources thereby isolating processes for one another and enabling scaling up and out and ②resolving all dependencies for a microservice automatically and deploying them in the virtualized environment. To realize the first part, we need mechanisms that exist within the OS to construct an environment within which each process can only access and manipulate resources provided within this environment. Within the Unix OS, these mechanisms are called *chroot jail*, *capabilities*, and *namespaces*. Recall the the Unix filesystem has the hierarchical tree structure where the root directory is the root node of the filesystem tree, it is designated by the forward slash `/` and it is the parent directory of all other directories and files. By navigating to the root directory, a process can access other directories and explore their contents as long as the access control allows the process to do so. The Unix OS command `chroot` changes the root directory for a given process to some other directory. We discussed the use of jailing in the architecture of Mesos in Section 13.4. To remind the reader, it is equivalent to projecting a branch of the filesystem tree onto the visibility plane for a given process. For example, after executing the command `chroot /home/mydir /` we can see that we are in the root directory `/`. Unpleasantly, we also find out that we cannot execute any rudimentary Unix commands like `pwd` or `ls`, since they are not accessible from within the new virtual environment that we created with the command `chroot`. In fact, the process that runs in this new environment becomes "jailed" in it, hence the term *chroot jail*.

Consider the skeleton of a container management system that is shown in Figure 10.2. In line 1–3 necessary header files are included and the function `main`'s body is defined in lines 4–11. The program takes one command line parameter that specified the existing directory path in `argv[1]` and it switches to this directory in line 5. If the directory change is successful, then this directory is `chroot`ed in line 6. If the call is completed successfully, an interactive shell is started in line 8. The user may executed Unix commands in the shell and then type `exit` to finish the program. However, if the `chroot` call executes successfully, the shell will not start and the error code will be outputted "Failed: no such file or directory" in line 8. The reader can experiment with this program by commenting out line 6 and line 9. The reason that this error code is

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  main(int argc, char **argv) { //argv[0] contains the name of this program
5   if(chdir(argv[1])==0) {//argv[1] contains the name of the destination dir
6      if( chroot(".") == 0 ) {
7          execlp("/bin/sh", "/bin/sh", "-i", NULL);
8          perror("Failed: ")
9      }
10  }
11  return (EXIT_SUCCESS);   }
```

Figure 10.2: The skeleton C program of a container management system.

produced is that once the directory was `chroot` jailed, the directory `/bin` becomes inaccessible and the program `sh` cannot be accessed any more.

It leads us to the second part of the idea to resolve all program dependencies and move them to the `chroot`ed directory. One solution is to obtain the list of all dependencies by using the Unix command `ldd` that determines all shared objects that are required to run a specific program. Consider the output of this command in Figure 10.3. A technically savvy reader will notice that this command is executed under Cygwin, a Unix OS layer ported to the Windows OS. The command is shown in line 1 with the output of all shared objects/libraries shown in lines 2–14. The output consists of the name of the shared object with the separator `=>` followed by the absolute path to the shared object and its hexadecimal memory address in parentheses. All these dependencies and the program itself can be copied to the destination directory and the set of paths to these objects can be created to mimic the paths to the original destinations.

Question 7: Discuss the possibility of implementing `chroot`ing within a hypervisor type 1 to split the execution environments for different applications.

Detecting and recreating dependencies within `chroot` jail directories can be easily automated by extending the original skeleton program shown in Figure 10.2. Moreover, many popular platforms and frameworks have well-known and documented dependencies, so it is possible to organize these dependencies in compressed file archives and store them in repositories from which they can be easily retrieved on demand. For example, Java programs depend on the *Java Runtime Environment (JRE)*, so different versions of these environments can be retrieved and uncompressed in jails. Of course, some dependencies may be specified manually in configuration files.

The isolation provided by jailing containers is not absolute. Consider different levels of caches, e.g., L1, L2, and L3 caches. When the OS kernel context switches between different processes that run in different containers, the data in these caches may be shared between processes. A full discussion of the computer architecture issues with cache data sharing is beyond the scope of this book, however, depending on the type of memory address reference, i.e., physical or virtual, the OS reads the info from a special unit called *Translation Lookahead Buffer (TLB)* to determine what process

```
$ ldd /bin/sh
  ntdll.dll => /cygdrive/c/WINDOWS/SYSTEM32/ntdll.dll (0x7ffb3b1f0000)
  KERNEL32.DLL => /cygdrive/c/WINDOWS/System32/KERNEL32.DLL (0x7ffb3a300000)
  KERNELBASE.dll => /cygdrive/c/WINDOWS/System32/KERNELBASE.dll (0x7ffb37780000)
  ADVAPI32.DLL => /cygdrive/c/WINDOWS/System32/ADVAPI32.DLL (0x7ffb3a1f0000)
  msvcrt.dll => /cygdrive/c/WINDOWS/System32/msvcrt.dll (0x7ffb3a150000)
  sechost.dll => /cygdrive/c/WINDOWS/System32/sechost.dll (0x7ffb3a550000)
  RPCRT4.dll => /cygdrive/c/WINDOWS/System32/RPCRT4.dll (0x7ffb3b0c0000)
  SYSFER.DLL => /cygdrive/c/WINDOWS/System32/SYSFER.DLL (0x74ef0000)
  cygwin1.dll => /usr/bin/cygwin1.dll (0x180040000)
  cygreadline7.dll => /usr/bin/cygreadline7.dll (0x579cd0000)
  cygiconv-2.dll => /usr/bin/cygiconv-2.dll (0x5461d0000)
  cygintl-8.dll => /usr/bin/cygintl-8.dll (0x5ee2d0000)
  cygncursesw-10.dll => /usr/bin/cygncursesw-10.dll (0x48ca30000)
```

Figure 10.3: The dependencies of the program `/bin/sh` are obtained using the utility `ldd`.

and which memory address accessed by this process has the corresponding entry in the cache. Yet, the cache access happens in parallel with the TLB access, so that the speculative cache read will reduce the wait if the TLB lookup confirms the cache hit. This example shows that the absolute isolation is difficult to achieve, however, for most applications the level of isolations provided by containers is enough.

Two more important concepts enable effective program isolation and access management: *capabilities* and *namespaces*. A user who logged into the Unix OS with the superuser privileges has the permission to run any command and to perform any operation on any software or hardware installed on the system and to install any software package or to configure new hardware modules. However, only trusted system administrators have the superuser privileges and even if they do, they often log into the system as regular users to prevent accidental damage by executing, for example, the command `rm -r /root`. Capabilities are special permission bits that can be set programmatically for processes to execute some privileged commands and only for the duration of this privileged commands. For example, setting the capability `CAP_SYS_RAWIO` enables the process to access a USB drive without obtaining the superuser privileges. Fine-grained permission setting is the main benefits of using capabilities.

The reason that capabilities are important for containers is because the container owner is the superuser of the jailed environment, however, the owner is not the superuser of the Unix OS in which this container is deployed. It means that a process that runs in a container may request certain capabilities, e.g., to access ports or files in the system to obtain the data that is necessary for further execution. Consider a situation when a containerized process must `ssh` to a different computer to run a different program. On the one hand, `ssh` is an external dependency for this process, however, it is also a popular Unix utility. Instead of copying it to every container, it is easier to grant containerized processes capabilities to run this and similar utilities.

Finally, Unix namespaces partition resources by grouping isolating them from the exposure to other groups. Consider as an example the XML namespaces that are used for resolving name clashes for elements and attributes in an XML document. By specifying the scope of the namespace, the same name for elements and attributes can be

used as long as they exist in different namespaces. In Unix, namespaces is a relatively new addition and in general there are seven core namespaces:

Mount namespace isolates mount endpoints of the filesystem;

UTS namespace isolate hostname and domainname areas;

IPC namespace isolate interprocess communication mechanisms;

PID namespace isolate the process identifiers sets;

Network namespace isolate network interfaces;

User namespace isolate user and group identifier areas;

Cgroup namespace isolate cgroup root directory.

Consider a situation when a process is started in a container and the OS assignes it a unique identified called *Process ID (PID)*. Unremarkable as it is, this PID should be unique for the duration of the OS session that starts with booting up the OS and ends with its shutting down. When a new process is `forked`, the OS assigns it a unique PID. However, when users spawns new processes in their virtual containers, they view them as if they control the entire OS environment whereas they only own a part of it within the container. Thus, if the OS uses the same PID numbering scheme in each container, then there will be many processes with the same identifiers thereby violating the PID uniqueness property. Using PID namespaces for containers enables the OS to use the same PID assignment scheme in different namespaces thus preventing PID clashes. The same argument is applied to other namespaces.

> **Question 8**: Explain how to prevent name clashes for the filesystem that is used by different containers.

Finally, one unresolved problem is provisioning resources to programs that run in containers. Even though the users of containers are presented with the virtual OS environment and they are given the sense that they have the full control of the resources, the reality is that there are many containers running in the same OS environment that compete to access and control the underlying physical resources. To enable users to provision resources to their containers, some versions of Unix kernel offer the implementation of *control groups (cgroups)* to partition resources (e.g., memory, CPU time) to groups of processes. A container management system uses namespaces, control groups, capabilities, program dependencies, and `chroot` to create and manage lightweight VMs. i.e., containers within the single OS environment.

We conclude with an observation that containers can be deployed in other containers using nested recursive structure – a RESTful microservice can be installed in a web server container that is deployed in a jailed container that is deployed within a VM. Technically, it may be possible to run a VM in a container; however, a point of this gedanken experiment is to show the theoretical composability of container objects

```
1 FROM java:11
2 WORKDIR /
3 ADD PrintSomething.jar PrintSomething.jar
4 EXPOSE 8080
5 CMD java -jar PrintSomething.jar
```

Figure 10.4: An example of Dockerfile for creating a container with a Java program.

rather than indicate a practical need to do so. We moved from deploying distributed objects in VMs because of the high overhead of the latter – putting a container into a VM or vice versa must accomplish certain objectives that will outweigh the effect of using a container with an expensive overhead.

10.4 Docker

Docket is one of a popular container management systems that is an open-source project and in addition, its enterprise version is sold commercially. Docker uses the core ideas that we described in Section 10.3 to realize container management in `chroot` jailed environment with several other additional services. Similar to the definition of a process as an instance of a program, a Docker container is an instance of an *image*, which is recursively defined as a set of layered images each containing software modules that the target program depends on. For example, a RESTful microservice written in Java depends on the JDK and the Apache web server that in turn depend on the JVM that depends on Ubuntu Linux. These dependencies are specified in a simple declarative program in a file called `Docketfile` that the Docker platform uses to build an image that can be instantiated into a container on demand.

Consider how a container is created and deployed with a simple Java application that contains a single Java class named `PrintSomething` with the method `main` that outputs some constand string to console. Next, we create compile the program and create a `jar` file named `PrintSomething.jar` without the customized manifest data file. The program can be run with the command `java -jar PrintSomething.jar`. Next, we create the file `Dockerfile` that contains the following commands as it is shown in Figure 10.4. In line 1, we specify that the image `java:11` will be included in the container image. The first command in the file is `FROM <image>:<tag>` that specifies the name of the docker image and its tag, optionally, and in this case it is the image that contains the JVM and the JDK version `11`. The next command in line 2 is `WORKDIR` that sets the working directory for all commands that follow in the docker configuration file. In line 3 the command `ADD` copies the file `PrintSomething.jar` to the filesystem of the image. The command `EXPOSE` in line 4 specifies that the docker container listens at port `8080`. Only clients on the same network can contact the container on this port; to make it available to clients located in other networks, the command `PUBLISH` should be used. Finally, in line 5 the command `CMD` runs the deployed program in the container.

The container image can be built by executing the command `docker build -t PrintSomething`. Once the image is built successfully, it can be uploaded to

```
1 import docker
2 client = docker.from_env()
3 print client.containers.run("mycontainer", ["ls", "-la"])
4 for container in client.containers.list(): container.stop()
```

Figure 10.5: The skeleton C program of a container management system.

a Docker repository, e.g., DockerHub[3] using the docker command `docker push /PrintSomething:latest`. The user must log into DockerHub prior to executing this command and create the repository. Once the image is created and pushed into the repository successfully, it can be pulled from the repository using the command `docker pull /PrintSomething` on a computer where Docker is installed and then the container can be run using the command `docker run /PrintSomething`.

Question 9: Install Docker and create a container for a simple Java HelloWorld application that writes the message into a text file.

Docker is created using the client/server architecture where the Docker's `daemon` is the central component of the server. Docker daemon is responsible for the entire container management lifecycle where it controls creation and deployment of Docker images. Docker clients issue commands to the docker daemon using the program `docker` as the *Client-Level Interface (CLI)*. Moreover, docker clients communicate with the daemon using the RESTful API calls that are described in Docker's documentation[4]. Consider a Python program that is shown in Figure 10.5. It uses the Docker SDK API calls to create a container running a version of the Unix OS and to execute a command to list the content of some current directory. In line 1 the docker package is imported into the program environment and in line 2 the docker client is initialized. The variable `client` references the local docker container environment, and using this variable in line 3 we obtain a list of deployed container, select a container named `mycontainer` that runs a version of Unix OS and send the Unix command `ls -la` to obtain a list of files in the current directory in the container. Then, in line 4 we iterate through the list of containers and stop each container on this list. This example illustrates how it is possible to create programs that orchestrates containers.

Deploying containers in the cloud is a rudimentary exercise. Consider the deployment instructions for Docker containers on Amazon Web Services (AWS) cloud[5] or the steps for deploying Docker applications on Microsoft Azure Cloud[6]. Specific details vary, but the idea is the same as we illustrated above with creating and deploying a container image with a Java application. All major cloud vendors provide mechanisms for container deployment. A programmer must ensure that ports are open and computing nodes at which container images are deployed are up and running. Often, it is

[3] `https://hub.docker.com`

[4] `https://docs.docker.com/develop/sdk`

[5] `https://aws.amazon.com/getting-started/tutorials/deploy-docker-containers/`

[6] `https://docs.microsoft.com/en-us/azure/devops/pipelines/apps/cd/deploy-docker-webapp?view=vsts`

necessary to attach load balancers to groups or "swarms" of containers to distribute the load among them. Mostly, deploying container images in the cloud environment is a purely system administration task. Docker and several other companies and organizations established Open Container Initiative as part of the Linux Foundation to create "open industry standards around container formats and runtime[7]."

When applications are run in their respective containers, they are expected to change the state of the container, e.g., to create or modify data in files. Since some of these files may be shared by applications in other containers, Docker uses a strategy called copy-on-write (CoW) where the OS creates a copy of the data object that a containerized process needs to modify. One problem with CoW is that it may be slow and inefficient in terms of the size to store written data.

Question 10: Discuss pros and cons of using the ZFS for Docker.

Consider a CoW file system where data is organized in blocks. A filesystem structure called *inode* stores information about file metadata and pointers to the blocks in which the file data is stored. Suppose the number of block pointers in an inode is limited to some N pointers. If the size of a block is less than some predefined threshold then the inode points to this block directly, otherwise, the data block will contain a pointer of its own to point to the other blocks of data that contain the spillover data for the block. Various CoW filesystems have different constraints on how pointers and blocks are organized, but the idea is basically the same.

Now, suppose new data is written into a file or its existing data is modified. In CoW, the inode and blocks will be copied and new blocks appended to the pointers from the existing blocks. Depending on the granularity of a block and how copying is done, CoW may results in significant data duplication and worse resulting performance of the filesystem. Instead, it would make sense to use a filesystem that employ branch overlaying, where an immutable part of a filesystem stays in one layer and a mutable part of it is located in the other layer in its branch that extends some branch of the immutable layer. Essentially, branches are unions of the different filesystems where entries are either shown in all branches or they are hidden or combined using different techniques. This strategy is implemented using the *Union File System (UnionFS)* that combines disjoint file systems into a single representation by overlaying their files and directories in *branches*. With UnionFS, applications can save the local states of containers in the corresponding branches without changing globally shared data among different containers.

10.5 Kubernetes

The goal of Google's cluster management systems, Borg and Omega is to manage executions of jobs in a cloud computing cluster, we will discuss them in Section 12.4.. With the containerization of applications, the unit of deployment shifts to a container

[7] `https://www.opencontainers.org`

rather than a process or a sequence of tasks. A container does not only provide an isolation environment for a program – it contains images that resolve all dependencies required for the contained application to run. A key benefit is a new unit of abstraction for controlling and manipulating diverse applications in the heterogeneous cloud environment – a container!

The container abstraction is important for three main reasons. First, it is important to know in the cloud environment if the running application is responsive. Naturally, each application may expose a friendly interface to response to a heartbeat signal with some statistics of its execution. However, enforcing developers of each application to implement such an interface is very difficult. Since containers automatically provide a RESTful interface for monitoring its content, the monitoring and health check problem is addressed without imposing any constraints on the deployed applications. That is, the container itself is an application; the actual running processes within the container are encapsulated by its boundaries and hidden from the cloud that is only concerned with the measured properties of the deployed containers.

Second reason is in obtaining information about the structure of the deployed application and its dependencies. Knowing that the application uses some library and a database that already exist in the cloud enables the cloud to automatically optimize the space and the execution time for deployed containers by positioning them on the server where the library already exists and scheduling them to use an existing connection pool to the given database. Since metadata that describes the content of a container is given apriori, this data is embedded into the built container and it shares this data with the cloud management system.

Finally, the third reason is in changing the load balancing using composability of containers. Instead of thinking about individual hardware resources, they are abstracted as container units (e.g., a container with two 3GHz CPUs and 8GB of RAM). Within these outer container units, inner containers are deployed to use hardware resources attached to the outer units called *pods*. When an application is divided into collaborating microservices, they can be deployed in inner containers in a pod and resources can be scheduled among them based on their constraints (e.g., log data collection can be performed periodically and sent to a log store somewhere on a server in the cloud).

> **Question 11**: Discuss the analogy of optimizing the space and resources with physical shipping containers vs the software containers.

Kubernetes is an open-source container-management system that was developed internally in Google before it was released as an open-source project [32]. Kubernetes website contains a wealth of information and there are many tutorials on how to set up and use it[8]. In this section, we will give a high-level architecture overview of Kubernetes and highlight its main functionality for scheduling and executing jobs in the cloud environment.

A key concept in Kubernetes is a *pod*, as we described above, it is an execution context that contains one or more containers and resources to execute software applications

[8] `https://kubernetes.io/docs/concepts/workloads/pods/pod`

located in these containers. The execution context of a pod is shared by the containers hosted in this pod, and these containers may in turn have their own contexts to provide additional levels of isolation. Of course, the reason that containers are co-located in one pod is that they are logically and physically coupled, i.e., the applications in these containers in a pod would otherwise be executed on the same computer or on computers linked on the same network. All containers within the same pod are assigned the same IP address and they have direct access to the shared context of the pod. The applications within the same pod can communicate using *Interprocess Communication Mechanisms (IPC)* like pipes or shared memory. Therefore, one main benefit of putting containers in the same pod is to improve their communication latency.

Kubernetes controls the life cycle of a pod. Once a pod is created, it is given a unique identifier, and then it is assigned to a computing node on which its applications execute. The pod can be terminated due to the node becoming inoperable or because a timeout is reached. Unlike VMs, pods cannot be migrated to nodes; a new identical pod is created with a new unique identifier and it can replace the existing pod that will be terminated after the new pod is assigned to a different node. Pods can be controlled via exposed REST interfaces that are detailed in the documentation[9].

Question 12: Explain why pods are not migrated to different nodes.

Kubernetes is created using client/server architecture where nodes are clients which are connected to the kubernetes master (or just the master) that uses distributed key/-value storage called `etcd`[10]. The master is implemented as a wrapper called `Hypercube` that starts three main components: the scheduler `kube-scheduler`, cloud controller and manager `kube-controller-manager`, and the API server `kube-apiserver`. Kubernetes offers CLI via its command `kubectl`. Each node runs a process called `kubelet` that serves as a client to the master and it runs a web server to respond to REST API calls. Kubelets collect and report status information and perform container operations. Kubernetes proxy `kube-proxy` configures IP table rules upon startup and it load balances jobs for each container. It also acts as a network proxy by maintaining IP table rules to control distribution of the network packets across the containers.

The job of using Kubernetes for a cloud datacenter is a system administration job. Its essential components is to understand the network topology of the data center, its resources, and how to create scripts that effectively distribute containers across the clusters and efficiently deploy them to maximize the performance. Whereas an application developer cares how to create, build, and debug applications and deploy them in containers, a cluster system administrator has the goal of deploying these containers in pods and scheduling these pods to nodes to utilize the cloud efficiently. This high specialization of jobs within the cloud datacenter is a relatively new phenomenon.

[9] `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.12/#pod-v1-core`

[10] `https://coreos.com/etcd/`

10.6 Application Container Specification and rkt

App Container is an open-source project for defining rules and constraints for application containerization and it is governed under the Apache 2.0 license[11][12]. *App Container Specification (appc)* provides a generic blueprint for defining, packaging, distributing, and deploying portable and self-sufficient containers uniformly. The specification defines the structure of the container image, its discovery mechanism and process, the pod structure, and the runtime. Since we discussed pods in Section 10.5, it is useful to study how pods are represented in `appc`.

A pod manifest is a declarative file that is divided into nine sections that specify the applications deployed in the pod, mount points, filesystem statuses, ports, volumes, annotations, and isolators. An example of an isolator is the memory handle and its limit to a certain threshold. A pod manifest is ingested by the container management system at startup, processed and exposed via its metadata service to enable clients to obtain information about deployed pods at runtime.

Multiple implementations of `appc` exist – `Jetpack` is an implementation of the specification for FreeBSD and `Kurma` is an implementation for Linux. In this section, we briefly review `rkt` – an implementation of the specification for Linux whose architecture offers three main advantages over Docker. First, `rkt` does not use the daemon configuration and it means that running containers do not have to be suspended or stopped when the container management system is updated and the daemon has to be shut down and restarted. Second, the communications between `rkt` components are done using the filesystem – a generic and well-explored OS mechanism that enables flexibility and low-latency access.

Just like other OSes, Unix OS has a concept of a special program called a daemon, which runs in the background mostly autonomously with little interactions with users if any. There are many tutorials on writing Unix daemons whose core is reduced to spawning a process by using the system call `fork` and setting output streams, current directory, and file masks correctly. Unfortunately, the reality is a bit more complicated when it comes to industry-strength deployment and maintenance when the stability of the entire system and its performance are paramount concerns.

> **Question 13**: List main benefits and drawbacks of creating an application as a microservice rather than a daemon.

First of all, many installations of the various flavors of Unix have many daemon programs. Essentially, the boot system of many flavors of the Unix OS is designed around `init` – the main daemon process that starts up when Unix boots and it is assigned `PID = 1`. It is the ultimate parent process and all other processes are spawned by `init`. A newer module is `systemd`, a set of Unix utilities that replace the classic service `init`. In many version, the main daemon process consults a configuration file that specifies what other daemon programs it should spawn. Examples of well-known daemon programs are `crond` and `nsf`, a job scheduler for tasks that need to

[11] `https://github.com/appc/spec`
[12] `https://github.com/appc/spec/blob/master/SPEC.md`

be executed at some time intervals and the service for the network file system. Having many daemon programs may make the system less stable, since crashing and restarting services make the system resources partially unavailable.

Second, many daemons work in the sleep/wake up/poll/work pattern where the daemon program sleeps for some time, then wakes up, polls some resources and perform the task. Doing so introduces an unnecessary overhead on the system, since the polling mechanism consumes resources without performing any useful tasks. Unlike Docker that has its own daemon implementation, `rkt` does not have its own daemons, it depends on `systemd` to start tasks. For example, to start a container, the following command is executed: `systemd-run --slice=machine rkt run coreos.com/etcd:vX.X.X` where `X` designates a (sub)version of the key/value store `etcd`. Interested readers can study the documentation that describes how `rkt` is used with `systemd`.

10.7 Summary

In this chapter we learned about cloud containerization. We started off by understanding the concept of microservices and how they need lightweight runtime environments, since they are small and they run for a short period of time. We moved on to review web servers as containers for RESTful microservices and showed how programming language support is important for writing concise and easy-to-understand code of microservices. Next, we introduced the container concept, where it provides a closure environment for fully packaged services. We rolled our sleeves and showed how containers can be implemented from scratch using basic OS services. Interestingly, this chapter shows how close the systems and OS research is intervowen with cloud computing. To enable some desired behavior, it is often required not to look much father away than the API calls exported by the underlying layers of software. The last three sections of this chapter review three popular industry-wide systems: Docker, Kubernetes, and the application container specification and its implementation, `rkt`. It is noteworthy that this is a fast-evolving field and the readers should check technical news sources and documentation to keep abreast with new development in containerization and microservice deployment platforms.

Chapter 11

Cloud Infrastructure: Organization and Utilization

Composing distributed objects in VMs into large-scale applications is heavily influenced by the organization of the underlying cloud infrastructure. After all, not all application's objects are equal: some of them contain large amounts of data and respond to RPC calls of their clients by sending subsets of this data; other objects communicate with databases by retrieving data, caching them, and providing this data to their clients; others operate on small amounts of data but run computationally intensive algorithms; and there are many variations of services provided by distributed objects. Whereas data retrieval-oriented objects should be hosted by VMs that are running on computers that are connected to the fastest networks, computationally intensive objects should be given a large number of powerful vCPUs on demand. Therefore, it is important to know the physical cloud infrastructure and its components and how to utilize them to improve the performance of the cloud-based applications.

11.1 Key Characteristics of Distributed Applications

We have learned that creating distributed applications is more difficult than monolithic applications due in part to separate memory spaces between clients and servers. Even when function calls are made in the same address space they incur some overhead or *latency*, a term that designates the period of delay when one component of a computing system is waiting for an action to be executed by another component [1]. The function call latency in the same address space includes resolving function parameters into memory addresses, creating a frame on the stack, pushing data for function paramters onto the stack, passing the function address to the CPU, maintaining the stack bookeeping, and obtaining return values and jumping back to the address of the instruction that follows the function call. Even though we assume that this latency is zero, it is not, and for a distributed application it is much bigger, since processes communicate by sending messages. Latency matters because many cloud-based distributed applications have SLAs that dictate the thresholds on response times, and if the threshold is not

met, the computation is considered failed, pretty much the same way as if the result of the computation was incorrect. In fact, for a number of cloud-based applications a slightly incorrect result may be more acceptable than the result that was computed with a significant latency.

> **Question 1**: Give examples of applications where incorrect results are acceptable. Explain how the incorrectness of a result can be measured in terms of some proximity measure to the correct result.

In a distributed application its components exchange messages in parallel, i.e., executing two or more activities at the same time, and resources are accessed and manipulated via message passing, often asynchronously. When two or more messages access and manipulate the same resource simultaneously, they may violate the state of the resource by interleaving instructions triggered by messages in an incorrect way. For example, *races* occur when the order of instructions that set values of some object determines the eventual state of this object. A change in the order of the instruction is triggered by the order of messages that access and manipulate the object. To ensure that this concurrent access does not result in an incorrect state, synchronization algorithms and techniques are used that may linearize the accesses by queuing messages and applying them to the resource in a sequential order. Doing so increases the latency of the application, since messages will stay in the queue for some time before being processed. When communications among processes take unpredictable time and the speed of running each process depends not only on its instructions and data but also on messages from other processes that access resources concurrently, it is very difficult to predict the behavior and performance of the distributed application.

In order to avoid burdening programmers with low-level implementation details, various techniques, libraries, and VMs are used hide the complexities of the distributed architecture from programmers by providing transparent view of certain aspects. We have already seen how RPC provides access transparency, where functions are called the same way in client applications regardless whether the server is local or remote, thus hiding the physical separation from clients. Also, the clients are oblivious to the location of the remote RPC server, since they refer to the remote objects by their logical names that are mapped to physical locations, hence location transparency. If the latency of one location increases beyond some threshold, the remote object may be moved to a new location while the client can continue to make remote calls to this object, thus using migration transparency. Hiding parallelism and concurrency enables clients to send messages to remote objects without worrying about timing these messages among one another. In fact, by replicating an object, it is possible to assign messages to replicas to process them concurrently, but the cost will be incurred later to synchronize the states of these replicas. Providing all these transparencies comes at a cost – different types of hardware in the cloud infrastructure exist that have different impact on the latency and transparencies.

A key characteristic of a distributed application is that it is not possible to obtain a snapshot of its consistent global state, in general, that includes the content of channels, when processes of the application continue to execute at their own speeds and

they send and receive messages asynchronously and the channels between processes contains messages with which these processes communicate [34, 66]. Also, communication times are bounded, so an indefinite latency does not exist. Without the ability to obtain a consistent global state at a given time, when a partial failure occurs when some computer or a part of a network malfunctions, it is very difficult to determine its effect on the correctness and the performance of the distributed application in general. Masking failures can be done in computer hardware, so it is important to understand the cost and impact associated with the choice of hardware for the cloud infrastructure to provide failure transparency.

Question 2: How are failures masked in the map/reduce computation?

Up to this point, we abstracted away the physical components of the cloud infrastructure, because we concentrated mostly on programming interfaces of the distributed objects. However, knowing the physical organization of the cloud infrastructure enables stakeholders to assign distributed components to the physical resources that can decrease the latency and it can improve the overall performance of the application significantly. Consider the following example where the cluster, A is located in North America and the cluster E is located in Europe. Suppose that a data object is hosted in A and its client object that is hosted in E, also hosts some data. The client pulls the data from A and merges it with its local data using some algorithm. Suppose that there is client objects located in Middle East, M, that submit requests to E to obtain a subset of the results of the data merging using some criterion. A question is whether the assignment of objects to certain clusters results in the a better performance of this distributed application with higher utilization of the provisioned resources?

One may argue that we need to know more about the sizes of the objects in A and E. If A contains big data and E has a small database, then transferring the large chunks of data from A to E results in a very high latency, whereas transfering the data from E to A can be done with little overhead. Of course, one can argue that the object from E can be moved to A, however, there may be two arguments against doing it. First, since the M client is located close to E, the network latency may be much smaller when obtaining the results from E rather than A. Second, privacy rules and other country-specific laws may dictate that certain types of data and computations must not leave specific geographies. On the other hand, A may not have computational capacity to perform intensive computations provided by E's infrastructure. Thus, we can see from this simple example that the cloud infrastructure makes a significant impact on the application's architecture as well as its deployment to achieve the best performance.

In this chapter, we will review various elements of datacenters and how they are organized to host cloud-based applications. At an abstract level, we view the organization of a cloud as a graph, where nodes are processing units and edges are network links using which these processing units exchange data. The nodes can represent a whole datacenter, a cluster in a datacenter, a computer in a cluster, a CPU or a Graphic Processing Unit (GPU), a VM, or any other resource that is used to execute instructions. All elements of the graph have latencies and costs and their capacities are provisioned to distributed applications. We will review latencies of various components of the cloud

infrastructure, then we will briefly review major components, their functions, and discuss their advantages and drawbacks.

11.2 Latencies

Certain latencies are important to know[1], since they drastically affect the performance of the cloud-based application. Using approximate ranges for latency values and other parameters that include the cost of devices, we will construct a hierarchy of devices and their aggregates in the cloud infrastructure. Since big data applications have significant presence in cloud computing, we will construct a storage hierarchy of devices that host and transmit data for computations.

We start with the finest granularity of devices, specifically chips on the CPU that are used as caches. Accessing L1 cache has the smallest latency in the storage hierarchy, since it is directly built into the CPU with the zero wait state, where the CPU accesses the memory without any delay. In general, to access memory the CPU must place its address on the bus and wait for a response, so zero wait means that there is no wait, which is fast but also expensive. A part of L1 keeps data and the other part keeps program instructions. L1 is the most expensive and the smallest available storage usually less than 100Kb, whose reference takes between one and two nanosecond or 10^{-9} second (for L1 cache hit), and it is the smallest latency time in the latency hierarchy. L2 cache is located on a chip that is external to the CPU, it larger than L1, able to hold around 512Kb and even over one megabyte on some CPUs, and it is also a zero-wait data access. The latency of L2 cache hit is approximately three to five nanoseconds, and the idea is that if accessing L1 cache results in a miss, then L2 is accessed to check if the data is there before accessing the main memory.

Given how fast L1 cache is the reader can wonder why it is not made much bigger and why it is so expensive. Increasing the size of L1 cache means increasing the area of the CPU, and larger die sizes decrease yield, a term that refers to the number of CPUs that can be produced from source materials. Suppose there are D defects in a square inch of the source material. If each chip takes S square inches, the likelihood is *DS* that a chip will have a defect. The higher the likelihood is, the lower the yield of the chip production, and this is one of the reason why circuit sizes decrease. The other reason is the amount of heat that is often proportional to the size of the chip, and when bigger chips produce too much heat, the computer systems overhead and get damaged sooner and bulkier cooling systems are needed to keep these computers functioning. These and other reasons make these small fast memories very valuable and expensive.

L3 caches frequently appear on multicore CPUs, and unlike L1 and L2 caches, which are exclusive to each CPU core, L3 caches are shared among all cores and even other devices, e.g., graphic cards. The size of an L3 cache can be 10Mb and they are frequently used to store data from inter-core communications. The latency of the L3 cache hit is in the range of 10-20ns depending on the CPU. Finally, L4 cache is large measuring in 100+Mb and it has the access latency after L3 cache miss in the range of 30-40ns. L4 is also called a victim cache for L3, since the data that is evicted from

[1] `https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

L3 is moved into L4, which can be accessed both by the CPU and the GPU, making it potentially valuable for high-performance graphic applications.

When the instructions are executed by the CPU and a jump instruction takes a branch that was not predicted by a branch prediction algorithm that runs within the CPU, then the prediction is discarded and the proper instruction from the correct branch is loaded. This latency is in the range of three nanoseconds, which is comparable with L2 cache hit. This is approximately the latency of a local function call if we count simply locating the address of the first instruction of the function. Recall a context switch between threads and processes, when a quantum expires or a thread/process is blocked by I/O and the OS kernel saves the state of one thread/process, puts it to sleep, and restores the state of some other thread/process to execute it. Measuring them is a nontrivial exercise in general, and the cost of a context switch with Intel CPUs is between 3,000 and 5,000ns at the high end. Depending on a benchmark the context switch latency for a process is approximately 1,500ns, although these numbers may vary depending on the number of processes/threads and available CPUs[2]. Interestingly, transferring data between the CPU and the GPU is expensive and it depends on the size of the data and the types of the processors, and the range is from 5,000ns for 16 bytes to 500,000,000ns for 64Mb [55].

Question 3: If adding more VMs to a server will lead to a higher overhead with context switches and putting these VMs on separate servers will lead to the higher network latency overhead, how would you design an optimal approach to distribute the VMs?

Mutex (un)lock takes approximately 20ns, which is close to L3 cache hit and it is five times faster than referencing a location in the RAM (i.e., $\approx$100ns), which is 20 times faster than compressing 1Kb using the Zippy compressor (i.e., $\approx$2,000ns or 2μs). Sending 2Kb of data within the same rack network takes less than 100ns, which is more than 50 times faster than reading one megabyte of data from the RAM sequentially (i.e., 5,000ns or 5μs), which is three times faster than a random read of a *solid-state drive (SSD)* a flash memory technology (e.g., over 15,000ns or 15μs). A round trip of a packet in the same datacenter takes 500,000 ns or or 500μs. A seek operation of a *hard disk drive (HDD)* with a moving actuator arm takes six times longer, three milliseconds, and three times longer than reading 1Mb from the same HDD sequentially. While the cost of the SSD is much higher, its performance w.r.t. sequential and random reads and writes beats the HDD with a moving arm actuator by several orders of magnitude. Finally, a packet roundtrip from CA to the Netherlands takes approximately 150 milliseconds.

To summarize, the storage/latency/cost hierarchy can be viewed as the following chain where the latency numbers are given in parentheses in nanoseconds: L1(1)→L2(3) →L3(10)→L4(30)→Access RAM(100)→Rack Send2K(100)→ RAM Read1Mb(5,000) →SSD Read1Mb(15,000)→Datacenter Send(500,000)→ HDD Read1Mb(1,000,000)→ USB SSD Read1Mb(2,000,000) → HDD Seek(3,000,000) → USB HDD Read1Mb

[2] `http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html`

(20,000,000)→SendCA2N(150,000,000). These numbers are approximate and they depend on the type of benchmarks used to collect these measurements. A fundamental question is how to exploit the hierarchy when designing cloud-based applications to minimize the cost and maximize the performance of the applications.

```
1 int array[10000][10000];
2 for(int i = 0;i < 10000; ++ ){for( int j = 0; j < 10000; j++) {
3         array[i][j]++;   }    }
```

Figure 11.1: C program fragment for iterating through the values of the two-dimensional array.

We can make a few conclusions based on these latencies. L1–L4 caches are very valuable, however, programmers are often forbidden from using them directly and instead algorithms are used in the OS to load caches often based on data locality. That is, if an instruction accesses a byte in a RAM, it is likely that contiguous bytes will be accessed immediately after this instruction. Consider an example of a C program fragment in Figure 11.1. The representation of the two-dimensional array is contiguous in the RAM where the first row 10,000 integers are laid out followed by the second row of 10,000 integers and so on. OS cache locality algorithms will preload a few thousand contiguosly laid out numbers into L1 and the execution of this program will take, say, 100ms depending on the CPU. Now, suppose that the indexes are jumbled up in line 3. Even though the programs are semantically equivalent, the execution time will increase more than tenfold because of the frequent cache misses, since accessing columns means non-sequential accesses of the contiguous memory where the next data item may not be loaded in L1 cache. The compiler GCC offers instruction `__builtin_prefetch`, however, programmers are discouraged to use it, since it is very difficult to determine how to optimize the code by loading some of its data into caches, especially when executing in a multithreaded environment. However, structuring data in a way to preserve the locality is a good advice to improve the performance by reducing cache misses.

Question 4: Would it make sense to build a compiler that analyzes all components of the application from the performance point of view using the values of latencies and optimize the allocation of the components to different VMs in the cloud environment? Please elaborate your answer.

Next, it is much faster to load all data directly into the RAM and then perform computations on the data, since seek and read operations on HDDs with moving arms is very expensive even compared with the corresponding SSD read and write operations. However, RAM is quite expensive and many commodity computers have limitations on how much RAM can be installed. For example, as of 2017, one terabyte of RAM costs approximately $5,000, which is an expensive proposition for a data center that hosts tens of thousand of computers. And big data sets are often measured in hundreds of terabytes, which precludes loading all data into the RAM. Interested readers can read more about computer memory and caches elsewhere [46, 70].

Next comes a permanent storage such as HDDs and SSDs. HDDs have large capacities, they are cheap, but they are also slow. SSDs have lower capacity and the

higher cost than HDDs, but they are much faster in terms of the seek time and the read and write operations. They are also prone to failure – various statistics exist for diffent brands of drives showing the annualized failure rate up to 35%[3]. A study done in 2007 shows that HDD replacement rates exceed 13% [133], which concurs with the Google study that shows that at the extreme every tenth HDD in the datacenter can fail once a year [121]. The authors of the study also concluded the following: "In the lower and middle temperature ranges, higher temperatures are not associated with higher failure rates. This is a fairly surprising result, which could indicate that data center or server designers have more freedom than previously thought when setting operating temperatures for equipment that contains disk drives."

Since it is highly likely that an HDD can fail in a datacenter, to prevent data loss, data should be replicated across multiple independent computers. One implementation of this solution is to use a *Redundant Array of Inexpensive Disks (RAID)*, which we will review in Section 11.4. The other is to use custom solutions for creating multiple replicas of data, which is expensive and these replicas should be synchronized. Clients will enjoy replication transparency and failure transparency, since with the correctly implemented solutions, they will see only one failure-free data object. Consider the replication and failure transparencies in the map/reduce model, where the computation and the data are discarded on the failed computer, the shard is replicated to some other healthy computer and the worker program is restarted there. Deciding how much data is to load into RAM from a hard drive and at what point in program execution is one of the most difficult problems of performance engineering.

Question 5: Discuss cost-benefits of using RAID versus replicating shards on multiple computers and handling failures by restarting computations.

Making a remote call in the cloud infrastructure means that a network latency is added to the cache, RAM, or disk latency. A network latency is smaller if remote calls are made between servers that are mounted on the same *rack*, which is a sturdy steel-based framework that contains multiple mounting slots (i.e., bays) that hold blade or some other rack-mounted servers that are secured in the rack with screws or clamps. Blade servers designate a server architecture that houses multiple server modules (i.e., blades) in a single chassis. Each rack may house its own power supply and a cooling mechanism as part of its management systems, which include a network router and a storage unit. *Hyperconverged infrastructures* contain server blades that includes computational units (e.g., the CPU, the GPU), local storage, and hypervisors, which can be controlled by issuing commands using a console installed on the rack for managing all of its resources. Due to the locality of all servers on a single rack and a router or a switch that connects these servers in a close-proximity network, remote calls among these rack-based servers incur smaller latency when compared to datacenter-wide calls or remote calls among servers that are located in different datacenters. We discuss networking infrastructure in datacenters below.

[3] `https://www.backblaze.com/blog/hard-drive-failure-rates-q1-2017`

11.3 Cloud Computing With Graphics Processing Units

A few large scale cloud businesses introduced clouds based on *Graphics Processing Unit (GPU)* for high-performance computational tasks, e.g., machine learning and scientific computing[4][5]. *General-purpose computing on GPUs (GPGPU)* is becoming increasingly popular, where GPUs, which are built for graphics computations, are used for traditionally CPU-bound computing (e.g., sorting or encrypting data). In this section, we will learn about GPUs and how they can be used in cloud computing.

A fundamental difference between the CPU and the GPU is that the CPU is based on *Single Instruction Single Data (SISD)* model and the GPU is based on *Single Instruction Multiple Data (SIMD)* stream model [51] [120]. In SISD the CPU executes one instruction at a time on a single data element that is loaded from a storage into the memory prior to executing the instruction. In contrast, an SIMD processor (i.e., the GPU) comprises many processing units that simultaneously execute instructions from a single instruction stream on multiple data streams, one per processing unit thereby achieving a higher level of parallelism than the CPU.

While the computer memory is considered a one-dimensional array for the CPU, the GPU is designed for graphic computations where the memory is viewed as a two-dimensional array. This distinction is dictated by using two-dimensional screens to output data that is computed by the GPU. Because of this difference in the memory models, it is required that data be transformed to fit target memory layouts when transferred between the CPU and the GPU.

> **Question 6**: Write a formula for translating one-dimensional array into a two-dimensional one.

A high-level view of the architecture of the GPU is shown in Figure 11.2 with the on-chip GPU-portion of the architecture encompassed by the dashed line. Arrows represent data transfers between the components of the architecture. We describe this architecture from a viewpoint of offloading a unit of computation from the CPU to the GPU, so that we understand the latencies of communications between the CPU and the GPU as well as the benefits of parallel computing offered by the GPU.

Once this computation is invoked at the CPU, a request is made to transfer the data upon which should be encrypted and decrypted from the CPU memory to the GPU. A specially dedicated hardware card (e.g., PCI express card[6]) reads data from the CPU memory in blocks of some predefined size and transfers this data to the read-only memory of the GPU (also called *texture*). The texture is designed as a two-dimensional array of data, and it can thought of as a representation of the screen with each element of the memory mapped to some pixel of the screen.

The GPU comprises many processors $\mathtt{P}_1, \ldots, \mathtt{P}_n$, and these processors invoke functions (also called *fragments*) on the input data that is stored in the read-only texture.

[4] `https://cloud.google.com/gpu/`
[5] `https://aws.amazon.com/hpc/`
[6] http://en.wikipedia.org/wiki/PCI_Express

Modern GPUs have dozens of processors. Since the GPU is a type of the SIMD architecture, all processors execute the same fragment at a given time. Programmers can control the GPU by creating fragments and supplying the input data. However, programmers cannot control the scheduling mechanism of the GPU, and therefore they often do not know how data items are scheduled into different pipelines for the processors. It is assumed that the computation is split among different processors which execute the fragment in parallel and independent of each other, achieving a high efficiency in processing input data.

A goal to achieve high parallelism is the reason for having separate read-only and write-only textures. If different processors could read from and write to the same texture simultaneously, the GPU would have to have a lot of additional logic to prevent reading from a previously modified position in the texture memory in order to improve parallelism and prevent errors. Having this logic as part of the GPU architecture would worsen its performance, however, the downside of having two separate textures is that it is more difficult to design programs for the GPU.

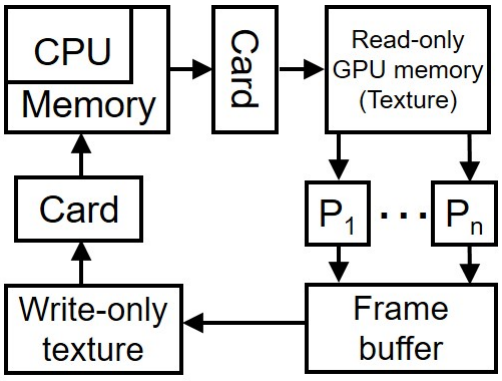

Figure 11.2: A high-level view of the GPU architecture and interactions with the CPU.

When the data in stored in the read-only texture, the call `draw` is invoked to execute fragments on the GPU. This call is equivalent to the call `exec` which creates a process in Unix, and it has a high overhead [138]. The call `draw` loads and initializes data and fragments and passes control to the processors of the GPU to execute instructions of the fragments. In general, it is better to execute fewer calls `draw` on larger data blocks to improve the performance of the GPU [137]. Of course, the performance of the GPU depends also on how much instructions and data are independent of one another.

Once processors executed fragments on all input data in parallel, they write results into the frame buffer. The data from the frame buffer can be displayed on the screen, or they can be passed back to the CPU. To do that, the data is written to the write-only texture and then transferred to the CPU memory using a hardware card. This card may be the same as the card used to transfer data from the CPU memory to the GPU texture if this card supports bidirectional transfer. In general, transferring data between the CPU and the GPU is fast and scalable. Increasing the size of the data blocks leads to the linear increase in the transfer time. The results of the old experiment with `GeForce 8800 GTX/PCI/SSE2` show that the data transfer rate is about `4Gb/s` [85], which is more than ten times faster than many types of internal hard drives. However, certain CPU latency is involved when initiating a transfer of a block of data [120].

The GPU programming model is well-known and documented. Programmers can use standard libraries (e.g., OpenGL, DirectX, or Cuda) to write applications for the

GPU. These libraries hide hardware-specific details from programmers thereby improving design and maintenance of GPU-bound programs. Since, the GPU inherently supports parallel computations, many parallelizable computationally intensive algorithms are good candidates to run on GPU clouds.

11.4 RAID Architectures

Since hard drives can fail frequently with the complete loss of their data, additional protection is needed to prevent the loss of the data in case of failure. To reduce the chance of losing data, *redundant arrays of inexpensive disks (RAID)* combine two or more single hard drives with algorithms that replicate the data across these hard drives [113] [26, p.103]. Since it is unlikely that two or more combined drives will fail at the same time, the data will not be lost if one drive fails. The other drive will still hold the data, and provided that the first drive is replaced quickly after the failure, the data from the healthy drive will be replicated on the newly installed drive that replaced the failed one. This is the essence of RAID.

Question 7: Is it possible to completely lose data stored in a RAID system?

RAID is classified into six levels and a number of hybrid combinations of these levels to achieve various guarantees of performance and data protection.

RAID 0 : the data on a hard drive is segmented and segments are distributed across multiple hard drive in RAID using a *round-robin* algorithm where each data segment is assigned to a hard drive in a circular order. This technique is called *data striping* and its biggest advantage for RAID is in increasing the speed of data access, since multiple segments can be written and read in parallel from multiple drives. However, a failure of one drive results in losing data segments stored on this drive, which means that RAID 0 cannot be used to prevent data loss completely.

RAID 1 : the data is mirrored on multiple hard drives, thereby decreasing the possibility that the data is completely lost if at least one drive remains healthy when the others fail. However, writes are a bit slower because data must be mirrored to all drives, whereas the speed of reading data is not affected and is the same as the one of RAID 0.

RAID 10 : contains four hard drives organized in pairs; data is striped across the pairs and then mirrored within each pair.

RAID 2 is not currently used in any major system due to a number of reasons including the complexity of the *error correction code (ECC)* and the need for all hard drives to spin in perfect synchrony, also called a *lockstep*, when all operations occur at multiple systems in parallel at the same time. The data is striped at the bit level and distributed along hard drives with a parity code for each segment

stored on separate hard drives, which makes it inefficient. The data transfer rates are roughly proportional to the number of the ECC hard drives, which means that the cost of RAID 2 will increase as the transfer rates are increased. A simplified version of a parity code is adding the bit one when the number of bits in the segment is even and bit zero if the number of bits in the segment is odd. RAID 2 uses Hamming ECC, which is computationally intensive and we will not review it here. Interested readers can learn more about ECC from various textbooks on this subject [75, 101].

To understand how ECCs work in RAID at a very high level, consider the logical XOR operation that we designate it here with the following symbol $\oplus$. Two equations that describe the meaning of XOR are the following: $x \oplus 0 = x$ and $x \oplus 1 = \neg x$, where x is a variable that represents a bit. From these equations it follows that $x \oplus x = 0$. Now, suppose that there are two segments of bits that we designate with the variables a and b and we store these segments on two separate hard drives in RAID 2, and the product $ecc = a \oplus b$ is stored on a third hard drive. Without the loss of generality, suppose that the drive that holds the segment a failed. After replacing it with a new hard drive, the value of a will be restored by computing $ecc \oplus b \equiv a \oplus b \oplus b \equiv a \oplus 0 \equiv a$.

RAID 3 is also based on data striping, which is frequently done on the byte level, and it computes the parity code that is stored on separate hard drives. Hard drives that store data should also be synchronized and their number must be a power of two due to the use of parity code for error correction. Since all strip writes are done simultaneously on multiple drives, the write performance should be high, but unfortunately, it is limited by the need to compute the parity code.

RAID 4 is similar to RAID 3 with the difference of dividing data in blocks from 16Kb to 128Kb and the parity code is written on a separate hard drive, thus creating contention during concurrent writes.

RAID 5 is similar to RAID 4 with the difference of writing parity codes to multiple hard drives, called distributed parity. Therefore, write performance is increased.

RAID 50 : the data is striped across groups of hard drives that are organized in RAID 5 architecture, thus significantly improving writes performance and adding fault tolerance to RAID 0.

RAID 53 or RAID 03 : the data is striped across groups of hard drives that use ECCs instead of mirroring for fault tolerance.

RAID 6 is similar to RAID 5 with the difference of adding another parity code block.

RAID 60 : the data is striped across groups of hard drives with distributed two-parity ECCs used for fault tolerance.

RAID 100 is a striped RAID 10 with a minimum of eight hard drives.

RAID architectures are more expensive, since they require more hard disks and more elaborate design of disk controllers. In addition, the performance versus the

cost tradeoffs are complicated, since they depend on how applications use the I/O – an application that frequently reads the data, computes results, and writes small data infrequently needs a different type of RAID when compared with an application that writes large data chunks frequently. However, in many cases it depends on the cost of computation – in the map/reduce model a single computation on a shard can be repeated in a case of a hard drive failure, thus obviating the need for using a RAID architecture. Purchasing commodity computers in a datacenter with a specific RAID architecture will put certain application at disadvantage with respect to the performance; purchasing computers with different RAID architectures will increase the level of heterogeneity, making the hypervisors decide what specific computer with a matching RAID architecture should an application be deployed on. Often, commodity computers are purchased without any RAID configuration, only dedicated computers may host one if customers specifically request them.

11.5 MAID Architectures

Non-RAID storage architectures refer to as collection of hard drives that do not cooperate in providing a level of fault tolerance. Main non-RAID architectures include *Just a Bunch of Disks (JBoD)*, *SPAN*ning volumes, and *Write Once Read Often (WORO)* arrays of hard drives, which we will collectively call *Massive Array of Inexpensive Disks (MAID)* [37]. A main idea behind MAID architectures is to create large volume storage that is viewed as a single hard disk from the client's perspective, but it is assembled from many single hard disks. That is, the view of continuous storage that spans multiple hard drives is implemented by a software layer that translates client's requests for accessing large files into concrete addresses to chunks of data that are spread across multiple hard drives, like we saw it with the GFS.

> **Question 8**: Suppose we replicate data chunks on different MAID drives. Compare this level of drive failure mitigation with the RAID systems.

The need comes from requirements for storing and accessing big data – many companies and organizations need to store petabytes and exabytes of data nowadays, and in the next decade breaking the size barrier to store and process zettabytes of data. Magnetic tapes are cheap and they can store up to 200Tb of data on a single tape[7], whereas hard disk, whereas the largest hard drive capacity is approximately up to four times smaller. Also, give the number of hard drives to provide the same capacity as a single magnetic tape, it takes up to ten times more cost to power the RAID-based hard disks than the magnetic tape. On the other hand, accessing data on a magnetic tape is sequential and very slow. Therefore, MAID architectures enables users to have both the storage capacity and the high speed of accessing the data.

Unlike RAID architectures, the main goal of MAID is to reduce the energy consumption while maximizing the amount of available storage and guaranteeing good

[7] `https://www.engadget.com/2014/04/30/sony-185tb-data-tape`

performance. MAID architectures use caching and workload prediction to improve performance. The other element is a level of abstraction – JBoD exposes individual hard drives. Users decide how to partition the data across these multiple independent hard drives. In a way, this is the simplest and least useful implementation of MAID. However, studies show that most data is read-only, with over 50% of written data accessed only once or not accessed at all. Splitting this data across multiple hard drives in JBoD increases the speed of reads, since they can be done in parallel.

A more sophisticated MAID architecture involves spanning or concatenating hard drives where they are organized in a linked list with the last block of data on one hard drive pointing to the first block of data on some other hard drive. Windows OS uses a drive extender module that presents a single logical name space to hide the MAID complexity, allowing users to add physical hard drives to computers without changing the logical structure of the file system (i.e., no new hard drive logical volumes appear in the file system after adding a new drive). In datacenters, network-attached storage (NAS) can be used, where hard drives are implemented as computer appliances with network cards that enable NAS components to easily integrate with MAID architectures to expand storage. *Storage area networks (SAN)* provide block-level data storage that is organized in its own network, and separate software layer must be build to provide an abstraction to users that they access locally attached hard drives with filesystems, whereas NAS is directly attached to a computer that integrates the storage into its filesystem. Both SAN and NAS can be used in MAID to provide the appearance of a single large storage and to replicate distributed objects for increasing reliability and availability of cloud-based applications.

11.6 Networking Servers in Datacenters

In cloud computing datacenters, distributed objects are deployed on different physical commodity computers (servers) that are located in different locations of the same datacenter or in different datacenters, and they communicate using messages, which are sent over the network. In this section, we will briefly review various networking infrastructure elements, how they are organized in network topologies, and how they are used in cloud infrastructures, and how they affect communication latencies. Interested readers are advised to read excellent book on cloud networking [89].

Question 9: Recall the OSI seven layer model. Map the infrastructure components in this section to the layers of the OSI model.

Recall that servers are attached to racks that are connected using a top-level switch, which forwards messages between servers on the rack and to servers on the other racks in the datacenter. According to different sources, a micro-datacenter houses less than 500 racks, whereas a standard size datacenter can house from 30,000 to more than 100,000 servers in up to 10,000 racks[8]. Google more than a dozen data centers around

[8] `http://www.netcraftsmen.com/data-center-of-the-future-how-many-servers`

the world consuming approximately 0.01% of the total power capacity on the planet, and as of 2013 both Google and Microsoft each own over 1,000,000 servers[9]. Racks contain server sleds, which plug into a networking board in the rack shelf, and with sleds, some rack components can be upgraded without major disruption to other components. Since multiple server sleds share network connections within the same rack, shorter cables are used and fewer NICs thus reducing the cost and complexity. Organizing these servers and racks of them into an efficiently networked cloud computing facility is a fundamental problem of cloud computing.

At a physical connection level, a network interface device (e.g., NIC or motherboard LAN) connects servers to the data center network via an Ethernet rack-based switch. A virtual switch, vSwitch is often used to connect servers on the rack[10], which is a shared memory switch, since the data often stays in the same shared memory on the servers and pointers to local address spaces of the servers are passed between VMs with subsequent translation by the hypervisors. Since many multitenant VMs often share the same physical server and the hypervisor, vSwitch enables them to bypass network communications, especially if the output of one VM is the input to the next VM, i.e., they are pipelined.

Question 10: Suppose that you can include the knowledge about the servers that run on the same rack into the hypervisors that run on these servers. Explain how the hypervisors can use this information to improve the performance of applications that are executed on these servers.

Two popular datacenter designs are called *top of rack (TOR)* and *end of row (EOR)* [89]. In TOR, servers are connected within the rack with short copper RJ45 patch cables, and they are plugged into the Ethernet switch that links the rack to the datacenter network with a fiber cable to aggregation Ethernet switches that are placed in some designated area of the dataveter. With every server connected by a dedicated link to the ToR switch, data is forwared to other servers in the rack, or out of the rack through high-bandwidth uplink ports. Since thousands of VMs can run on a single rack that can host 50+ servers and they can share the same network connection, 10Gb Ethernet links servers with the ToR switch. Thus, ToR is also a gateway between the servers and the rest of the datacenetr network. Since ToR intercepts all packets coming to or going out of the rack, it can tunnel and filter messages by inspecting packet headers and match various header fields using custom-programmed rules. Doing so reduces the complexity of administering the network on the datacenter scale.

Alternatively, in EoR design, racks with servers, usually a dozen of them or so are lined up side by side in a row with twisted category 6 pair copper cables connecting them via patch panels on the top of each rack. Each server is connected using short RJ45 copper patch cable to the corresponding patch panel. Bundle of copper cables from each rack are laid out either over the racks or under the floor. Unlike ToR where each rack has its own switch, EoR switch connects multiple racks with hundreds of

[9] `http://www.ieee802.org/3/bs/public/14_05/ghiasi_3bs_01b_0514.pdf`
[10] `http://openvswitch.org`

servers, thus simplifying updates for the entire row at once. ToR design extends the L2 network where *media access control (MAC)* addresses are used for message delivery in a spanning tree network layout whereas the EoR design extends a L1 cabling layer with fewer network nodes in the architecture of the datacenter.

Distance plays a significant role not only in the organization of the network design in a datacenter, but also in deploying distributed objects in cloud-based applications. Consider that sending a message over a fiber optic cable between continents takes less than 20,000,000 nanosenconds, a bit more with cable Internet, $\approx$ 30,000,000 nanoseconds, and a whopping 650,000,000 nanoseconds over the satellite [11]. A key limitation is the speed of light, which travels in an fiber optic cable at about 70% of its top speed. Therefore, using 1Gib Ethernet takes 10,000 nanoseconds to receive a 10,000 bits, whereas over a T1 link of 1.6 Mib per second the same task takes 6,477,000,000 nanoseconds. The rule of thumb is that the propagation delay in cables is about 4,760 nanoseconds per kilometer. On top of that, there are inherent router and switch latencies, which is the time to shift packets from an input to output ports. The constituents of the latencies include converting signals to bits and packets, buffering data, performing lookups to determine an ports based on packets' destination address using IP in L3 for a router or L2 for a switch, and forwarding the packets from port by converting the bits to electric signals. Network devices such as routers and switches and saturated network links introduce more latency because network devices wait thousands of nanoseconds before they place a message onto their output ports.

The formula for the overall latency for sending a message over the network is $L_{Net} = L_{Send} + L_{Rec} + \sum_{netdev} (t_{exec} + t_{forward} + t_{queue}) + \sum_{paths} t_{send}$, where L_{Net} is the total latency of the message, L_{Send} and L_{Rec} are the latencies of sending and receiving the message correspondingly at endpoints, $\sum_{netdev}$ is the summary of latencies at each network device (e.g., switches and routers) along the sending path, where each devices spends some time to execute some code that inspects a message and takes some actions, t_{exec}, time to forward a message to the next device, $t_{forward}$, and time to (de)queue a message if the processing capacity of the device is fully utilized, t_{queue}, In addition, latencies for transmitting messages as electrical signals across paths between devices are summarized, $\sum_{paths}$ and added to the total latency [81]. Various measurement methodologies exist for evaluating the overall latencies of different paths within and between datacenters to achieve the desired level of SLA.

Adding significant complexity to the network organization of the datacenter results in various unpredictable situation, one of which is called a *network meltdown*, when the network latency increases to infinity due to excessive traffic. A starting point in network meltdown is a *broadcast storm*, when a broadcast message, often multicast and not necessarily maliciously crafted, results in a large number of response messages, and each response message leads to more response messages, described as a snowball effect. A particular network meltdown is described based on the configuration of the network where two high-speed backbones are connected to different switches to which racks are connected [119, 74–76]. A problem was with the configuration of switches

[11] `https://www.lifewire.com/lag-on-computer-networks-and-online-817370`

that send and analyze *bridge protocol data unit (BPDU)* data messages within a LAN that uses a spanning tree (i.e., loop-free) protocol topology. BPDU packets contain information on ports, addresses, priorities and costs and they are used to "teach" switches about the network topology. If a BPDU packet is not received within a predefined time interval (e.g., 10 seconds), a switch sends a BPDU packet and goes into a leaning mode, where it forwards received packets. When other switches become busy and the learning mode switch forwards packets, the network may exhibit looping behavior with messages circulating among switches and multiplying over time resulting in full network saturation and its eventual meltdown. As a result, cloud computing services come to a complete halt and recovering a datacenter to the working state may takes hours or even days.

Question 11: Can a network meltdown be prevented by prohibiting multicast messages in a datacenter?

There is also the cost dimension to cloud computing, since not only computing resources are provisioned to customers on demand, but also customers pay for the provisioned computing resources. Adding tens of thousands of network devices to datacenters adds tens of millions of dollars to their cost, which is propagated to the customers who pay for their provisioned resources. Given the high cost of networking equipment that contains many features that cloud providers do not need in their datacenter, it is no wonder that cloud providers create their own customized networking devices rather than buying them off-the-shelf [12]. The Open Compute Project (OCP) is an organization created by Facebook to develop hardware solutions optimized for datacenters[13]. Whereas OCP advocates a solution where full racks are assembled from standalone components, rack-scale architecture is an open-source project started by Intel that advocates production of completely preconfigured and fully tested rack solutions that can be replaced as whole units on demand [14]. At this point, it is too early to say what solution provides the most optimal deployment both from the performance and from the economical point of views.

11.7 Summary

In this chapter, we discussed the cloud infrastructure and how it affects the availability and reliability of cloud-based applications. We reviewed key characteristics of distributed applications and discussed various latencies in-depth. Next, we considered RAID and MAID storage organizations, reviewed the architecture of the GPU, and analyzed the networking infrastructure of a cloud datacenter. We showed that choosing the right infrastructure and organization of computing units in a datacenter affects

[12] https://www.geekwire.com/2017/amazon-web-services-secret-weapon-custom-made-hardware-networ
[13] http://www.opencompute.org/about
[14] https://www.intel.com/content/dam/www/public/us/en/documents/guides/platform-hardware-design-guide.pdf

latencies, which worsen the performance characteristics of distributed applications deployed on the cloud infrastructure. In addition, the cost of the infrastructure should be minimized, so that provisioned resources cannot become very expensive. In the end of this chapter we discussed initiatives to create cheaper commodity off-the-shelf solutions for datacenters that provide low latencies and superb performances.

Chapter 12

Scheduling and Balancing the Load for Distributed Objects

Deciding what object to host on what servers in a datacenter is difficult. Clients submit requests to distributed objects hosted on cloud servers, and the these servers provide a quantifiable computing capacity to process these requests, which is called a *workload*. Computational tasks that are submitted by cloud customers result in creating workloads. We will use the terms computational task and a workload interchangeably in this chapter to designate the work that servers must accomplish.

In general, the term *workload* includes not only the static part of the input to the application (i.e., specific methods with the combination of values for their input parameters and configuration options), but also the dynamic part that comprises the number of requests that contain remote methods with input values submitted to the application per time unit and how this number changes as a function of time [99]. For example, a workloads can specify that the number of client's requests to a distributed object fluctuates periodically according to the following circular function: $y_i = \alpha \times \sin t + \beta$, where α is the number if method invocations in the workload, β is the constant shift, and t is the time. When workloads are described by a neat pattern like a sinusoid wave, they are easy to predict and subsequently, it is easy to proactively (de)provision resources based on the descriptions of these workloads. Unfortunately, the reality of cloud computing workloads is much more complicated.

Recall that cloud workloads are often characterized by *fast fluctuations* and *burstiness*, where the former designates a fast irregular growth and then a decline in the number of requests over a short period of time, and the latter means that many inputs occur together in bursts separated by lulls in which they do not occur [115]. If a workload rapidly changes, thus affecting an application's runtime behavior when a new resource was being provisioned before the workload changes, this resource may not be needed any more by the time it is initialized to maintain the desired performance of the application in response to the changed workload. Finding how to distribute workloads to objects to maximize the performance and to handle these workloads efficiently is one of the primary goals of cloud computing.

12.1 Types of Load Balancing

In general, load balancing means distributing the workload, W across S servers in a datacenter, so that each server, $s \in S$ is assigned the equal workload $w_s = \frac{W}{S}$, however, if each server has different computing powers, $p_s \in [0,1]$, then $W = \sum_s \frac{W}{S} p_s$. That is, we normalize the computing power of all servers in the datacenter to hold a value between zero and one, where the latter means the highest computing power among all servers in the datacenter and the former means no power, meaning that the server is not available or it does not exist. A perfect load balancer distributes workloads across servers, so that each server has exactly the same workload at all times. It is needless to say that achieving a perfect workload distribution can be done only in few very restricted cases.

> **Question 1**: Give an example of an application where you can achieve a perfect workload distribution.

There are two main types of balancing the load between servers: *static*, where prior information about the processing power, memory, and other parameters of the server is used and *dynamic*, where information about the servers and their environments is obtained at runtime. These types of load balancing are each realized in seven models for which load balancing algorithms are created.

Centralized model centers on a controller that keeps information about all servers. Using this model makes sense for a smaller private cloud when the all servers are documented and few tasks with predefined performance requirements and resource demands need to be distributed across these servers.

Distributed models are more suitable for unpredictably changing cloud computing environments. Opposite to the centralized model, in the distributed model each computing node makes load balancing decisions autonomously, sometimes in cooperation with some other computing nodes. If a computing node has a high workload, it may initiate requests to other nodes to share a part of this workload, i.e., a *sender-initiated load balancing*, whereas if it has a low workload, it may ask other nodes to offload some tasks to it, i.e., *receiver-initiated load balancing*.

Preemptive model allows interrupting load balancing algorithms based on obtained measurements from the cloud and changing the steps of load balancing based on these newly obtained measurements. For example, if some mappers crashed in the map/reduce model, a preemptive load-balancing algorithm may be interrupted to schedule replacements to complete the mapping tasks.

Delay-free models requires tasks assigned to servers as soon as they are launched by customers, whereas in the (see the next item)

Batch model, tasks are grouped depending on some predefined criteria, e.g., deep-learning tasks may be grouped to be sent to the GPU-based servers.

Single model addresses load balancing of a task that is independent of all other tasks, i.e., its execution is not synchronous with the executions of other tasks, and opposite to it in the

Collective model accounts for dependencies among tasks. A pipe-and-filter task organization is a good example, where the execution of some task can proceed only if the execution of the previous tasks in the pipeline is finished. For example, a reducer worker can be scheduled only after some mappers complete their jobs in the map/reduce model.

One aspect of load balancing is that in a multitenant runtime environment where multiple VMs are run on a multicore CPU, one VM may process much higher workloads than the other VMs. In this case, some workload is "stolen" from the CPU's cores that execute the high-workload VM and it is shared with the cores that execute low-workload VMs. This type of load distribution is called *load sharing*, where shared workload is not necessarily balanced among computing nodes. An example of load sharing that we discuss below is migration of a VM from a highly loaded server to a lower loaded server without stopping the execution of processes inside the VM.

12.2 Selected Algorithms for Load Balancing

Load balancing numerous tasks across many servers can be reformulated as scheduling these tasks at the servers. Scheduling theory is a mathematical study of creating schedules with three major goals: quick turnaround time to complete the task, timeliness of a schedule where completion of tasks is conformed to given deadlines, and maximization of throughput that measures the amount of work completed during some period of time. Whereas some scheduling algorithms may run for a prolonged period of time to produce results for strategically important tasks (e.g., for a batch job of scheduling flights for an airline in a datacenter), many scheduling algorithms must finish quickly. Think of a scheduling algorithm within the kernel of an OS that decides which process to execute at a given time to satisfy multiple constraints (e.g., thread priority, resource usage). If the OS on a desktop computer is interrupted by a scheduler even for a second to make scheduling decisions, such behavior would not be acceptable to its customers. Thus, achieving these goals requires a trade-off between their various aspects.

> **Question 2**: Given an example of a situation where a scheduling algorithm may be allowed to run longer to produce a more precise schedule.

These goals translate into three main criteria for evaluating a load balancing algorithm: its complexity, termination, and stability. A complex algorithm is difficult to analyze and it may take a long time to compute the distribution of tasks to different servers. Executing a load balancing algorithm is not free, since its execution takes time and resources. Its overhead should be very small, preferably less than one tenth of a percent of the total execution time of the server and its resources and it should make scheduling decisions within this time period that are better than a random assignment

of tasks to servers. Also, a load balancing algorithm should terminate eventually by producing assignments of tasks to servers at specific time intervals. The time and resources that it takes to achieve the scheduling decision can be used as measures of the complexity of the algorithm. And the produced scheduling decision should result in the stability, a term that defined a divergence of the computed loads on servers from the average as time progresses. An unstable load balancing algorithm schedules tasks to servers only to lead to even worse distribution of loads, which would result in triggering this algorithm again and again to reschedule tasks ad infinitum.

Main components of a load balancing algorithm include task assignment rules (e.g., utilization of resources on a computing node, expected time to finish for a task), location of the server, and various constraints (e.g., not to trigger load balancing within some time interval since the last load balancing decision or to trigger load balancing only if certain state change occurs like reduction of the size of the database below some threshold value). In some cases, decisions are made based on internal resource utilization, for example, how much CPU or the memory is used on a server versus the utilization of the network bandwidth. Load balancing algorithms are considered fast if they can compute a decision within the time that is proportional to the number of servers among which the load is balanced. Of course, the absolute time that the scheduling algorithm takes is important and should be very small. The cost of a load balancing algorithm comes mostly from communications among servers about exchanging workloads, however, the cost is supposed to be offset by the benefits of better utilization of resources and the overall increase of efficiency of the datacenter the performances of the applications deployed on this datacenter.

> **Question 3**: How does the network latency affect the precision of a scheduling algorithm? Discuss how scheduling algorithms can be improved with network latencies taken into consideration when computing an optimal schedule.

We start off by listing non-systematic algorithms and techniques that do not take into consideration the semantics of tasks and existing workloads of servers. A key element of these algorithms is that if they are repeated for the same tasks, they are likely to assign them differently to the servers.

Random algorithm is a static algorithm that is the fastest and simplest to implement. Essentially, it consists of a random number generator that produces a value between zero and the total number of servers in the datacenter. Each server is assigned a unique integer between zero and some maximum value. When a task arrives, the random generator produces a number that designates a server to which this task is assigned. Random algorithm is the fastest, since it contains a single step of generating a random integer. Since random numbers are distributed and non-repeating frequently, it is highly likely that the distribution of tasks across multiple servers will result in close-to-equal distribution of workloads. Same idea is applied for selecting a task for execution on the same server. However, random algorithms do not use any information about the servers' capacities, the approximate durations of task executions, and the existing workloads on servers.

It is possible that multiple tasks can be assigned to the same servers while the other servers have no tasks to execute. Often, randomization is used in conjunction with other heuristics for distributing tasks to servers.

Round robin scheduling algorithms also belong to the class of static load balancing algorithms where each task on a server will be given an equal interval of time and an equal share of all resources in sequence. Dynamic round robin algorithms are based on the idea of collecting and using runtime information about tasks, servers, and the network to compute weights as real numbers between zero and one and use these numbers to multiply the time intervals for each task. Essentially, an implementation of a round-robin algorithm is a loop in which each tasks is executed until the time interval expired and then the other task from the list is selected for the next loop iteration. The main benefits of the round-robin algorithms are that they are fast, simple to reason about and to implement, however, as with random algorithms, little information about the tasks and the servers is used to make decisions.

Min-min and max-min algorithms take the input of the tasks sorted by their completion times. In both algorithms, a status table for executing tasks is maintained. The idea of the min-min algorithms is to give the highest priority to the tasks with the shortest completion time, whereas max-min algorithm first selects the tasks with the maximum completion time. Once a task is assigned, its status is updated in the table. In the min-min algorithm longer executing tasks wait longer to be scheduled, since the priority is given to shorter tasks. Conversely, in max-min algorithm, the longer tasks are scheduled before the shorter tasks. Each algorithm has its benefits, however, a common drawback is that the discriminated tasks may wait for a long time resulting in a serious imbalance.

FIFO load balancing technique schedules tasks for execution on the first come first serve basis. It is a fast and simple technique that suffers from the same drawbacks of random algorithms.

Hashing algorithms are diverse, but the main idea is the same – using some attribute of a task (e.g., the name of the program that is executed for a task), its hash value is computed and the task is assigned to the server whose name or IP address is hashed to the same value, e.g., $H_t \equiv_s |S|$, where H_t is the hash code, $|S|$ is the number of servers, and $\equiv_s$ is the sign of modulo operation. In a way, using a hash function is similar to a random assignment, since using the name of a program or the IP address of the server does not amount to using some knowledge about the behaviour of the task or the optimality of assignment of this task to a given server.

Diffusion load balancing algorithm (DLBA) is fundamental static systematic algorithm [40]. In DLBA, the network is viewed as a *hypercube*, a graph with n nodes and $D = \log_2 n$ dimensions, where each node has the total number of D edges. For example, a hypercube with one node has the dimension zero; with two nodes connected by the edge, the dimension of the graph is one, and with four nodes connected with four edges to form a square, the dimension is two and so on. Each node is labeled with its

order number and a constraint is that nodes are connected with an edge if their order numbers differ only in one bit. The number of bits in the order number is equal to the dimension with high-order padding bits set to zero. One can think of nodes as servers and edges as network links in a datacenter.

Question 4: Discuss how to use generic algorithms for load balancing.

A key step of DBLA is that at some time, t two neighboring nodes, u and v in a hypercube exchange a fraction of the differences in their workloads, $\frac{|w_u^t - w_v^t|}{D+1}$ that is inversely proportional to the degree of the hypercube. Only the information about the previous time step is remembered, i.e., $w_u^t = w_u^{t-1} + \sum_{v \in N} \frac{|w_u^t - w_v^t|}{D+1}$, where N is the set of all neighboring nodes for the node, u. One can visualize the process by seeing the workloads diffuse from some nodes to other nodes in the hypercube, one step at a time. Discussing it in depth is beyond the scope of this book, and interested readers can study the original papers that discusses constraints for achieving stability and a large number of subsequent diffusing algorithms published in the past three decades.

12.3 Load Balancing for Virtual Clusters

A *virtual cluster*, $V_C = \{M, B, N\}$, is an abstraction of a virtual network that connects VMs, M to an SDN, N at some minimal bandwidth, B [128]. Recall that each VM in a V_C hosts some software stack with application that communicates via a *virtual Network Interface Controller (vNICs)*, a device that connects a computing node to the network by receiving and transmitting message from and to routers at L1-L4 OSI layers, and virtual routers with other applications hosted in VMs over the WAN. These virtual clusters can be plug-and-play, since they can be easily connected with other clusters to quickly create a *virtual datacenter*, which is designed to solve large-scale problems.

Each V_C is embedded, i.e., its Ms are mapped to physical servers connected by the underlying physical network, $E_{V_C} : M \rightarrow S$, where S is a physical server. An optimal embedding minimizes the overall cost of the allocated servers and the capacities of the physical network links are utilized equally and at the same rate. A general problem of a virtual cluster embedding is to find a valid embedding at a minimal cost: $P_S \sum_{v \in V_C} C(M_v \mapsto S) + B \sum_{e \in E_{V_C}} L_e$, where P_C designates the computing power of the physical node, S to which an M is mapped, C is a cost function, and L_e is a physical network link for a virtual edge, e.

Recall in our cloud computing models that some VM accept requests from clients over the network and many other VMs communicate with one another internally over the datacenter network. When customers pay for renting VMs to run their applications, this time often does not account for the cost of sending messages over the network. As a result, embedding a virtual cluster into a datacenter requires reasoning about the network traffic in addition to the power of individual computing nodes.

The difference is in trying to allocate equal workloads to servers regardless of their computing power, or to allocate workloads proportional to the computing powers of

the servers. In this definition, we do not consider the work of moving data across the network to servers, which may be considered proportional to the network latency. However, not only workloads for specific tasks can be scheduled for objects assigned to different servers, but also network traffic should be taken into account when assigning tasks to run in VMs on servers.

Question 5: How does the size of the data that the distributed objects need to access affect the scheduling decisions for distributing workloads across these distributed objects?

In this section, we review a load-balancing algorithm called Oktopus developed jointly by researchers and engineers from Microsoft, Imperial College, London, and Cambridge University. Oktopus is shown in Figure 1 and it accounts for the network traffic for a specific tree topology in a datacenter, where VMs are organized in virtual clusters and hosted on servers, which are organized installed in racks, which are connected using TOR switches using cables links with some capacity, $\mathcal{C}$, to other switches that connect other racks with the root of the tree being the top parent switch [18]. A question is how to assign tasks to VMs by not only distributing them equally with respect to workloads but also by not consuming the network bandwidth more than $\mathcal{C}$.

A key idea of the algorithm Oktopus is twofold: first, each server has a predefined number of slots to run one VM in each slot, and each request contains not only the number of VMs, $\mathcal{N}$, but also the required bandwidth, $\mathcal{B}$, for each VM. Second, if $\mathcal{N}$ communicating VMs should be assigned to two servers, which are connected with a link and one server can host m VMs and the other server hosts $\mathcal{N}-m$ VMs, then the maximum bandwidth required between these servers is defined **`min(`**$(m \times B, \mathcal{N} - m) \times B)$. Indeed, the amount of data sent both ways is limited by the smallest bandwidth producer or consumer. Using these two ideas the algorithm computes the assignments of tasks to VMs greedily so that the tasks are distributed across available servers to satisfy the bandwidth constrains in the tree network topology of the datacenter.

Algorithm Oktopus is shown in Figure 1 in lines 1–17 and the helper function, Allocate, is given in lines 18–29. Given the number of empty slots on a server, $\mathcal{E}$, the number of VMs, $\mathcal{M}$ that can be allocated to a server, $\mathcal{L}$ with the remaining link capacity, $\mathcal{C}_{\mathcal{L}}$ is given in line 6. The number of machines with their total bandwidth should be less or equal than the remaining bandwidth, $\mathrm{C}_{\mathcal{L}}$. The algorithm starts with a physical server, $\mathcal{L}$, and it calculates the number of the VMs, $\mathcal{M}$, that can be assigned to this server to its open slots, $\mathcal{E}$. The the function Allocate is called, and it returns in the base case where it is applied to the server in line 19. However, if the level is higher, i.e., it is a switch, then for each subtree, t of the switch in loop 23–27, the number of VMs allocated to it, m can be the total number, $\mathcal{M}_t$, if enough slots are available, leaving the residual number of VMs to allocate, $m -$ **`count`** as well as the residual bandwidth.

The algorithm minimizes the message traffic at the top nodes in the tree, i.e., the traffic should be maximized within racks mostly and minimized across the datacenter. The choice is given to subtrees located at the same level of hierarchy, so that the traffic is directed using as few switches as possible.

Algorithm 1 `Oktopus` algorithm for load balancing in virtual clusters.

Inputs: Network tree topology Ω, Input Request $R :< \mathcal{N}, \mathcal{B} >$
$\mathcal{L} \leftarrow$ **GetLeafMachineNode**(Ω)
Terminate $\leftarrow$ **false**
while $\neg$ **Terminate do**
  for $\omega \in \mathcal{L}$ **do**
    $\mathcal{M} \leftarrow \{m =$ **min**$(\mathcal{E}, \mathcal{N})\}$ s.t. **min**$(m, \mathcal{N} - m) \times \mathcal{B} \leq \mathcal{C}_{\mathcal{L}}$
    if $\mathcal{N} \leq$ **max**$(\mathcal{M})$ **then**
      Allocate(R, ω, m)
      Terminate $\leftarrow$ **true**
    end if
    $\mathcal{L} \leftarrow \mathcal{L} + 1$
    if IsRootNode$(\mathcal{L}, \Omega)$ **then**
      Terminate $\leftarrow$ **true**
    end if
  end for
end while
return $\mathcal{M}$
function Allocate(R, ω, m)
if IsPhysicalServer(m, Ω) **then**
  return m
else
  count $\leftarrow 0$
  for $t \in \omega$ **do**
    if count $< m$ **then**
      count $\leftarrow$ **count** $+$ **Allocate**(R, t, **min**(m-**count**,**max**(M_t)))
    end if
  end for
end if
return count

12.4 Google's Borg and Omega

In this section, we discuss two Google resource management systems: Borg and Omega [32]. Borg is a *cluster management system* in which all computers in datacenters are partitioned in cells or clusters, where each cell contains around tens of thousand of computers and these cells are connected with a high-speed network [154]. Whereas the words *cluster* and *cell* can be used interchangeably, a cluster consists of one large cell in Borg and a few smaller cells that are used for testing or some special purpose operations. One computer in a cell serves as a central controller unit called the Borgmaster and each cell computer runs an agent process called Borglet, an instance of the client/server architecture pattern. Borgmaster runs the main process and the scheduler process, where the former serves client RPCs and the latter handles the internal job queue and assigns pending tasks to machines in the cell where each submitted

job consists of one or more tasks. Borglets connect their respective computers up to Borgmaster that polls these Borglets several times each minute to determine the states of the corresponding computers and push requests to them. Even though the Borgmaster represents a single point of failure for a cell, the multitude of cells makes the entire datacenter decentralized and less prone to the complete failure.

Borg handles two types of jobs: short latency-sensitive requests that last from microseconds to milliseconds (e.g., send an email or save a document) and batch jobs that may last from a dozen of seconds to days and weeks – sample Google cluster data are publically available[1]. Clearly, these types of jobs have different sensitivities to failures and the underlying hardware support – for example, shorter jobs do not run in VMs, since the cost of starting and maintaining a VM process is high. Instead, these jobs are run in lightweight containers that are explained in Section 10. Also, jobs may depend on one another or have specific constraints (e.g., to run on a GPU architecture).

The diversity of jobs creates serious difficulties for job scheduling and load balancing. Large jobs will take bigger computers and possibly, a smaller portion of their resources will be unused, which may be useful in case the workload increases when a job is executed. However, the resource usage becomes heavily fragmented. In contrast, trying to pack jobs tightly on a computer in a cell leaves little room for assigning more resources for jobs whose workload increase rapidly. And if the constraints are specified incorrectly for a job, its performance will suffer because no resources may be available. Finally, packing jobs tightly may lead to poorly balanced jobs in the cell.

Question 6: Compare Hadoop YARN scheduling with the Borg's scheduler.

Borg's scheduler addresses these problems by first determining which jobs are feasible given their constraints and available resources in the cell and then by assigning these jobs to computers using an internal scoring mechanism that uses a mixture of criteria to maximize the utilization of available resources while satisfying the constrains of the job. One interesting criteria is to assign jobs to computers that already have software packages on which the job depends. However, in the likely case where no computer is available to run a job, Borg will terminate lower priority jobs compared to the one that is waiting to run to free resources. As we will later learn, cloud spot markets use a similar methodology to preempt applications for which much lower fees are paid to run to enable more expensive applications to run in the cloud.

Omega is a cluster management system widely considered the next generation of Borg within Google [135]. Omega incorporated elements of Borg that were successfully used for years and added a new idea of sharing the state of the entire cell among many schedulers within each cell. After years of Borg deployment, the contours of two main types of jobs formed: the first major type is batch jobs with over 80%, however, the Borg scheduler allocated close to 80% of resources on average to service jobs that run longer than batch jobs and they have fewer tasks compared to up to thousands of tasks for each batch job. Scheduling a large number of jobs with many tasks optimally is a very difficult problem, and a single scheduler per cell takes dozens of seconds or longer to schedule a job.

[1] `https://ai.googleblog.com/2011/11/more-google-cluster-data.html`

In Omega, many schedulers working concurrently replace a single monolithic scheduler where each scheduler receives a shared state of the cell and it can make its own scheduling decisions. That is, each scheduler operates in parallel to other schedulers in a cell and scheduling decisions are formed by each scheduler as atomic transactions that will change the state of the cell by assigning jobs/tasks to resources. Naturally, many scheduling transactions conflict, since they will use the same resources. So-called gang scheduling works poorly where the entire transaction is committed atomically or not at all. Instead, incremental scheduling transactions are used in Omega, since scheduling non-conflicting operations first and resolving conflicting ones later enables the scheduler to avoid resource hoarding for large jobs and allow smaller jobs to use available resources in between the tasks of larger jobs.

12.5 VM Migration

As part of load balancing, a VM may be required to migrate to a different server to distribute the load more evenly or relocate it to a more powerful server to improve the performance. This is an important operation, since it requires computing resources and the network bandwidth. In addition, migration transparency requires that the application owners are not affected by migration of their VMs. Specifically, the application that is run within the VM should not be paused for a long period of time or terminated during migration. In fact, the cost of migration contributes to the cost of load balancing, so if the cost exceeds certain threshold level, the benefits of load balancing may be obliterated. In this section, we review three techniques for VM migration [82].

In *pre-copy* migration, the memory state of the source VM is copied to the destination VM while the program in the source VM continues its execution. Once copying is finished, newly updated memory pages within the source VM are identified and copied, repeating this process until there are no modified pages to copy or some threshold for the amount of copying is reached. Then, the context of the source VM (e.g., environment variables, files, open network connections) is transfered to the destination VM and the execution is resumed.

Pre-copy migration is popular and it is used in Xen [20] and vmWare [131], however, a biggest problem with it is when the application frequently writes into memory. As a result, the migration will use a significant amount of the available network bandwidth to transfer memory papers, prolonging migration time and thus making load balancing less effective to the point of the increased costs negating all benefits. In real-world, i.e., non-simulated cloud settings, 60 second latency means that the migration process failed [82].

The other technique for VM migration is called *post-copy*, where the VM context is copied from the source to the destination VM before the memory content is copied [74]. Next, the execution of the destination VM starts and in parallel, memory pages are being copied from the source to the destination VM. Ideally, memory pages that are needed sooner for the execution of the destination VM should be copied first, however, in general, it is not possible to determine which memory pages will be needed at what point in execution. Therefore, memory pages may be ordered for copying based on different criteria: by the most recent time they were accessed, by the

number of accesses, or by locality – start with copying the most recently accessed page and then copy other pages whose addresses are adjacent to the most recently copied pages. Since the source VM does not continue to execute, it means that the number of memory pages is bounded that should be copied to the destination VM.

Question 7: Discuss the design of a container migration algorithm.

A key problem with the post-copy technique is that when copying memory pages the VM executes an application that may need memory pages other than the ones being copied. If the requested page has not been copied yet, then an exception is created, the execution is suspended, the memory page copying is suspended, the required memory page is located in the source VM and transferred to the destination VM, after which the process resumes. Generating exceptions and interrupting the migration process is expensive, and if it happens frequently, the cost of migration will increase.

The third technique is called *guide-copy* migration that combines elements from the pre-copy and the post-copy migration techniques. The main idea is to transfer the VM context from the source VM the same way it is done in the post-copy technique, and then to continue to execute both the source and the destination VMs. The source VM becomes the guide context, and since the destination VM is likely to access the same memory pages as the guide context. Therefore, the guide context provides pages that the destination VM will need and these pages are copied in time for the access and exceptions are avoided mostly. However, if copying is delayed for any reason, then the exception is generated as in the post-copy technique. Disk changes are treated the same way as memory changes and disk pages are transfered when they are modified at the source VM. The technique may terminate the execution of the guide context at any time, since non-transfered pages will be requested later by throwing exceptions as it is done in the post-copy technique.

Executing both the source and the destination VM consumes additional resources, and the process of transferring memory pages should complete relatively fast to avoid long duplicated executions. However, it is possible that the application in the source VM may execute for a long time without accessing new memory pages, i.e., the accesses may happen in bursts followed by long periods of lull. This is the most unfavorable scenario for the guide-copy technique, which is handled by early termination of the source VM. Interested readers may obtain more details from the original paper [82].

12.6 Load Balancers Are Distributed Objects

As we have learned, the job of a load balancer is to distribute tasks across the datacenter to ensure close to equal distribution of the workloads. A simplified view of a load balancing algorithm is that it takes requests that represent tasks as its input and the output of this algorithm is the assignment of tasks to VMs. Suppose that VMs are already instantiated in the cloud and they accept HTTP requests that come to some URL at port 8080. We can view each HTTP request as an implicit call to a remote method of a distributed object that implements a load balancing interface. Once invoked, the

method will take an HTTP request as its input and forward it to one of the existing VMs that run the same application that processes these requests (e.g., a web store). Thus, a load balancer is a distributed object that works as a multiplexer to send requests to VMs to improve the workload distribution.

Question 8: Explain how you would design an adaptive system where schedulers are added and removed elastically depending on the inequality of distributing workloads across multiple servers.

Moreover, as a distributed object, we can write a program that as a distributed object measures the number of input requests, the utilization of existing resources, and if it determines that the workload distribution becomes too skewed in the datacenter, it can create a load balancing object on demand and assign input requests to it and the VMs which the load balancing objects will manage.

```
1  AmazonEC2Client awsEC2 =//obtain a reference to a client
2  AmazonElasticLoadBalancingClient elb=//obtain a ref to LB client
3  CreateLoadBalancerRequest lbRequest = new CreateLoadBalancerRequest();
4  lbRequest.setLoadBalancerName("lbcs441");
5  List<Listener> listeners = new ArrayList<Listener>(1);
6  listeners.add(new Listener("HTTP", 80, 80));
7  lbRequest.setListeners(listeners);
8  CreateLoadBalancerResult lbResult=elb.createLoadBalancer(lbRequest);
9  DescribeInstancesResult describeInstancesRequest = awsEC2.describeInstances();
10 List instanceId=//populate the list with VM instance ids
11 RegisterInstancesWithLoadBalancerRequest reg =
12         new RegisterInstancesWithLoadBalancerRequest();
13 reg.setLoadBalancerName("lbcs441");
14 register.setInstances(instanceId);
15 RegisterInstancesWithLoadBalancerResult regLBRes=
16                 elb.registerInstancesWithLoadBalancer(reg);
```

Figure 12.1: A Java-like pseudocode for using a load balancer with Amazon Cloud API.

A Java-like pseudocode for creating and using a load balancer with Amazon Cloud API is shown in Figure 12.1. The code is in general self-explanatory with the use of the Amazon API documentation. Once references to a Elastic Computing (EC2) client object are obtained in line 1 and to the elastic load balancer in line 2, a load balancer request object (i.e., the input) is created in line 3, named in line 4 and listeners for HTTP requests are created and attached to the load balancer in lines 5–7. A load balancer result (i.e., the output) object is created in line 8, and existing VM instances are obtained and registered with the load balancer in the remaining lines of the code. At this point, a load balancer start servicing inputs and directing them to the registered VMs. The flexibility of using cloud API to create load balancers and dynamically configure them enables programmers to reconfigure their architecture on demand to improve performance and scalability of their applications.

12.7 Summary

In this chapter, we synthesized the knowledge of VMs and cloud infrastructure to introduce the concept of load balancing. We discussed key characteristics of load balancing algorithms, their types, and reviewed a few basic non-systematic algorithms that do not take into consideration the load of the network. We also studied a fundamental diffusing load balancing algorithm and Oktopus, a greedy algorithm for datacenters that takes into consideration that the network load should be minimized. Next, we study techniques for migration of VMs that load balancers use to distribute the load more evenly. Finally, we discuss how to use load balancers as distributed objects to dynamically change architectures of applications in the cloud to adapt them to changing workloads.

Chapter 13

Big Data Processing with Apache Spark in Clouds

Recall that map/reduce is a programming model where a computing task is partitioned into mappers, which transform input data units called shards into key-value pairs, and reducers, which take these key-value pairs and compute the desired result. It is implicitly assumed that data are accumulated in sets (or batches) of shards before the map/reduce computation starts and they do not change for the duration of this computation. There is no interaction with users to obtain new data and the intermediate key-value pairs are persisted to facilitate fault recovery. As such, map/reduce computation is a perfect example of *batch processing*, which is perfectly suitable for running a long operation with a large batch of data and amortizing the overhead of data processing, which otherwise would be incurred when executing many operations over small subsets of the big data batch.

The latencies of program interactions with the HDDs impose significant overhead on applications that work with multiple datasets and also process real-time streaming data that arrive to the application via some open interprocess communication channel. Recall that reading one megabyte of data from RAM takes approximately five microseconds whereas reading it from an HDD takes approximately 1,000 microseconds or 200 times slower. Of course, in general, entire data sets are not written to an HDD and then re-read from it with every program's instruction, however, in the map/reduce computations writing intermediate results to HDDs is done frequently and it is a source of a significant run-time overhead.

A large class of algorithms can benefit from persisting data in memory and scheduling operations across available resources more efficiently. Consider an example of computing all subsets of the pairs of integers in a data set that sum to the same number. As a map/reduce job, this task can be translated into mapping the input stream of integers to key-value pairs with the key designating the sum and the string value designating the concatenated values of the summed integers separated with the comma. These key-value pairs then written into files, and then reducers retrieve them by reading data from these files. Alternatively, we can avoid this overhead by keeping the

data in memory, creating a hash table with the integer key that designates a sum of two integers and the value as a list of objects each holding the value of two integers whose sum is the one in the corresponding key. As new integers arrive in real-time, they will be directed to computational units that will compute their sum and add to a corresponding hash table, and these tables can be merged at a later time. Thus, these datasets can be reused repeatedly, unlike key-value pairs in map/reduce without the I/O overhead. That is in the nutshell the main rationale behind Spark, which extends the computational model of map/reduce with significant scheduling and performance improvements [160].

Consider as an example an algorithm for clustering a dataset based on detecting centers of data point clusters and grouping the data points together that lie within a certain proximity to the center of each cluster. This algorithm is implemented within an external loop whose condition is to determine whether the distance is smaller than some threshold value between the center of a cluster computed in the previous loop iteration and the newly computed center of the same cluster. Within this loop, the data points are grouped by the closest distance to the center of each cluster and then the centers of the grouped data point clusters are computed by averaging the distances between the data points to the centers of their respective clusters. The algorithm may take thousands of iterations to converge. Consider an implication of multiple iterations on the implementation of this algorithm using the map/reduce model. It means that after each iteration the output of the map/reduce program becomes an input to itself. The overhead of recording the outputs that will be discarded is significant. As an exercise, please implement this algorithm or one of its variations as a map/reduce application.

13.1 Spark Abstractions

A key abstraction in Spark is a notion of *Resilient Distributed Dataset (RDD)* that contains a memory-resident collection of objects – roughly equivalent to a shared dataset in the map/reduce model – that is divided into partitions from the original dataset that is located on a permanent storage using some sequence of steps. Each partition is assigned to a specific computing entity with resources that are sufficient to support the computational task on a given partition. The sequence of steps that divides the original dataset into partitions is a significant element of the RDD abstraction, since it enables the programmers to reason about operations on the entire dataset without thinking about how it is partitioned, where these partitions are located, and what resources are assigned to perform computational tasks on each partition.

> **Question 1**: To what degree does the map/reduce model allow programmers to abstract away the partitioning of the datasets?

Since the partitioned collection of objects resides in memory, it can be easily lost due to various reasons including electric power surge or some hardware failures. In this case, the RDD will be rebuilt later using the sequence of steps from the original dataset from the permanent storage (e.g., an HDD). Even though doing so will slow

down the computation, if failures like that happen infrequently, RDDs can speed up computations significantly by reducing the I/O overhead.

As an abstraction, RDD hides the implementation details that enable programmers to obtain a physical dataset and perform operations on them. This is a main reason why Spark does not define a particular permanent storage type and, unlike Hadoop, it does not have a filesystem that is associated with its datasets. Doing so makes Spark flexible to implement specifics of loading and saving RDDs – they can come from collections, files, databases, in short, from any dataset in any format that can be partitioned and transformed into some respresentation in the RAM.

Moreover, the Spark's flexibility of dealing with RDDs allows programmers to integrate real-time data streams in Spark processing, where a *data stream* is a potentially infinite collection of lazily computed data structures, i.e., the elements of the stream are evaluated on as-needed basis. Consider a stream of data from smart meters installed in houses in a city and how newly arrived sets of data can be obtained from a network connection and added to RDDs based on some partitioning rule. Doing so in Hadoop is nontrivial, since it requires modifications to the framework to add the data structures to the existing shards or to new shards, change the map/reduce program configuration to read from these newly modified shards to recompute the results. In Spark, new data structures are simply added to an RDD and the corresponding computation is triggered on it automatically. Programmers don't need to write any additional code for that.

Abstract parallel operations allow Spark programmers to define standard manipulations on RDDs. Key operations include *reduction*, which reduces the number of elements in a dataset to obtain its concise representation (e.g., multiplying all values in a list of integers reduces this list to the single value of the product of these integers); *map iterations*, which traverses all RDD elements and applies a user-defined function that maps each element to some value; the operation *collection* that gathers all elements of the RDD and sends it to the master node, also called the *driver* that allocates resources for Spark operations; and the operations *cache* and *save*, where the former leaves the RDD in the memory and the evaluation of its elements is done lazily, whereas the latter evaluates the RDD and writes it into some persistent storage. Thus, unlike the map/reduce model, the persistence is uncoupled from other operations thereby allowing the programmers to achieve greater flexibility and better performance in the design of the big data processing applications.

> **Question 2**: Is the RDD Spark abstraction applicable when processing the next element in a sequence-organized dataset depends on the results of processing the previous elements in the sequence?

Finally, Spark relaxes the requirement of data independence from the map/reduce model by introducing a concept of shared variables, i.e., it allows dependencies among data elements and enables certain synchronization mechanisms to use these data dependencies. Of course, allowing a free-for-all read and write access to shared memory regions can complicate the design and analysis of Spark-based programs to the point that scheduling them would not be effective, since a some components of the program may not utilize its resources because they may have to wait to obtain access to a shared

memory region. In many cases Spark applications are stateful, they need to keep the state and share it among the worker and the driver nodes. Yet, using a global memory region for sharing data is likely to lead to side effects that will make it difficult for Spark to enforce scheduling and resource distribution. Instead, the Spark introduces shared memory abstractions that restrict the freedom of accessing and manipulating the content of the shared memory.

Shared memory abstractions are called *broadcast* and *accumulator* variables. The former represents read-only data structures (e.g., an employee hash table that maps employee identifier to some file locations which contain employee data) and the latter designates storage that worker nodes can use to update some values (e.g., the number of words in text documents computed using the map/reduce model). An example of a broadcast variable is the following Scala statement `val bcVar = sparkContext.broadcast(List(1, 2))`, where the object `List` is sent to all worker nodes. However, when an associative operation like summation should be applied to a collection object to accumulate the result in a variable, it can be declared as an accumulator and used in mapping or reducing operations as the following examples: `val acVar = sparkContext.accumulator(0, "Counter")` and then `sparkContext.parallelize(List(1,2).map(_=> acVar+=_)`. The Spark infrastructure take care of enforcing the semantics of these sharing abstractions.

13.2 Data Stream Operations

One can view a data stream as a sequence of objects that are arriving to some endpoint. Consider the function `def endp(v:Int):Stream[Int]=v#::endp(v+1)` in Scala. When invoked, the function `endp(1)` recursively generates an infinite stream of integers starting with one – a function that does it is called *corecursive*. This endpoint `endp(1)` can be viewed as a port to which integers arrive infinitely. However, when streams are evaluated lazily, their elements actually appear only when instructions in client objects are executed that perform operations on these elements of the stream.

> **Question 3**: How can the data generated by a corecursive function be mapped to RDDs and scheduled for execution?

Next, we take a few elements, filter out those not divisible by three, and compute the sum: `endp(1).toList.filter(_%3 == 0).flatMap(x=>List(x, x*2, x*3).foldLeft(0)(_+_)`. The mapping operation, `flatMap`, takes the function as its parameter that transforms the list by expanding each of its element into a list containing the element itself, followed by the one multiplied by two, and then by three. Finally, the operation, `foldLeft`, takes in the accumulator value, zero in the parentheses, and a reduction function that sums the value of the accumulator designated by the first underscore with the next element of the list designated by the second underscore putting the resulting value into the accumulator and returning it. These instructions are pipelined in the execution chain with the output of the previous operation serving as an input to the next operation in the pipeline. One can view this pipeline as

a conveyer where the stream elements are diced, sliced, their values or their types are transformed into some resulting product.

Using this example, we can see that each operation produces a subset of the stream computed by the previous operation in the pipeline. Moreover, these operations can be performed independently on the subsets of the original stream from the endpoint, `endp(1)`. At a high level, Spark processing can be viewed as splitting the original steam into subset streams, which are set as RDDs, and then applying operations in pipelines to each subset RDD. At some points, the results of the pipelined operations on each RDD is aggregated and reported as the final computed result.

A key programming concept is *monad* that allows programmers to construct pipelines to process data in a series of steps. Monad is a function, `M[U]`→`(U`→`M[V])`→`M[V]`, where `M` is a type parameterized by some other types, `U` and `V`. One can think of the type, `M` as the container type, say a list or a set, and the types `U` and `V` are the types of elements that the container type hosts, say the types `Int` and `String` respectively. The function maps the parameterized type, `M[U]` or using our concrete example, a list of integers, `List[Int]` to the `M[V]` or a list of string, `List[String]` via the intermediate function that takes the values of the type, `U` and maps them to the values of the parameterized type, `M[V]`. With our example, one can view the input, `List[Int]`, as a container that holds employee unique identification number and the monad maps this input onto `List[String]`, a container that holds the names of the people who are assigned these identification numbers. These monadic functions enable type translations that are needed to construct pipelined operational sequences.

Consider the monadic function, `def flatMap[U](f:U=>M[V]):M[V]` that takes a function that maps the value of the input type, `U` to the output container, `M[V]` that contains values of some other type, `V`. In our example above, this function, `f` is represented as `x=>List(x, x*2, x*3)`. In fact, there are two functions: one that maps a value of some type, `U`, to a container that hold values of this type, `M[U]` and the other function that maps the container, `M[U]` to the same container that holds values of some other type, `M[V]`. The former function is defined `def unit[U](i:U):M[U]`. Applying the function `unit v` produces a list `v::Nil` where `::` is the concatenation operator that joins elements together into a list. The latter function, `bind` can be defined using the function `flatMap` as `bind container` ≡ `container.flatMap(f)`, where `f:U=>M[V]`. Thus, functions `unit` and `bind` are fundamental functions whose composition enables programmers to create data processing pipelined functions.

> **Question 4**: Write algorithms for bubble sort, quicksort, and k-means clustering using monadic functions in Scala.

Monadic laws govern the application of the functions `unit` and `bind`. The left-identity law stipulates that `unit(i).flatMap(f)`≡`f(i)`, since the function, `f` is applied to the input value, `i`. The right-identity law states that `c.flatMap(unit)`≡`c`, where the function `flatMap` applies the function `unit` to each element of the container, `c` resulting in the same container, `c`. Finally, the associativity law states that `c.flatMap(f).flatMap(g)`≡`c.flatMap(i=>f(i).flatMap(g))`, mean-

ing that the composition of the application of the functions `f` and `g` to the container object, `c` is equivalent to the application of the function `f` to each element in the container, `c` and producing some resulting element to which the function `flatMap(g)` is applied.

Monadic laws can be demonstrated using a simple example with `List` as the function `unit`. We have the function `def f(i:Int):List[Int]=List(i,i*2)` and the function `def g(i:Int):List[Int]=List(i+1,i+2)` that map the input integer to a list of integers where each element of the list is a transformed input integer. Let us apply `unit i` ≡`List(i)`. Then, we apply the first monadic law `unit(i).flatMap(f)`≡`f(i)` to determine that `unit(i).flatMap(f)`≡ `List(i).flatMap(f)=List(i,i*2)`, which in turn is equivalent to `f(i)= =List(i,i*2)`.

Appying the right-identity law, `c.flatMap(unit)`≡`c`, gives us the following result: `List(i).flatMap(List(_))=List(i)`≡`List(i)`. Finally, the associativity monadic law, `c.flatMap(f).flatMap(g)=List(i+1, i*2+2)`, and `List(i).flatMap(i=>f(i).flatMap(g))=List(i+1,i*2+2)`, so the law holds for this example.

The reason that monadic operations are important in Spark is because these operations can be chained together to construct a pipeline, which can be viewed as a conveyor with elements of the streams arriving in different shapes and in containers and robotic arms are monadic operations that filter, slice, dice, and generally chop and compress data in a variety of ways. One can view the computation of the average of the integer numbers in a list as a compression or reduction operation, since all integer numbers are "reduced" to a single floating point value. In order for robotic arms to process a rectangular object, it should be processed by some other robot prior to that to chop off parts of the round shape of this object to make it rectangular. Consider shapes of the objects as their types, and monadic operations become important to ensure the producer/consumer compatibility of the types for consecutive operations.

13.3 Spark Implementation

Spark is implemented using Scala, a JVM-based programming language with the mixed imperative and functional constructs. Scala comes with a rich library of monadic functions that programmers can use to process data using the functional programming model. More importantly, it enables flexible Spark program code analyses and manipulation by (de)serializing it using Java serialization API calls and transporting the code to the cluster nodes that will perform operations on the RDDs. Scala interpreter is tightly integrated with Spark to provide information to its components about the user program, so that it can be decomposed into constituent operations. Since Scala code is compiled into Java code, it can be executed on the generic JVM, thus enabling universality of Spark deployment on various platforms.

Question 5: Discuss pros and cons of compiling Spark code directly to the OS-level instructions.

Consider a Spark program in Figure 13.1 that obtains a list of integers separated by semicolons as streaming ASCII characters from a network address. In line 1 the Spark configuration object is obtained and it is used to line 2 to create a streaming context object where it batches the streaming data in ten second intervals. This stream can be viewed, for example, as a list of two strings, concretely expressed, for instance, as `List("123;5;6","123;1;579;16")`. The 10 second interval is not shown in the concrete representation where two text strings are combined in the list container. In lines 3 and 4, the object `streamContext` is used to create a socket that receives text stream at an address and a port designated in the variables `ipaddr` and `port` respectively. The input parameter `StorageLevel.MEMORY_AND_DISK_SER` specifies that the RDD is stored as serialized Java objects and the partitions that do not fit into the RAM will be stored to the HDD and they will not be recomputed when needed.

```
1 val sparkConf = new SparkConf().setAppName("SmartMeterReads")
2 val streamContext = new StreamingContext(sparkConf, 10.Seconds)
3 val intList = streamContext.socketTextStream(ipaddr, port,
4 StorageLevel.MEMORY_AND_DISK_SER).flatMap(_.split(";")).map(toInt)
5 val total = intList.map(_=>1).reduce(_+_).collect()
6 val err = intList.filter(_=>_< 0).map(_=>1).reduce(_+_).collect()
7 val sum = intList.filter(_=>_ > 0).reduce(_+_).collect()
8 val avg = sum/(total-err)
```

Figure 13.1: A Spark program for computing the average value for the stream of integers.

In line 4, two monadic functions are invoked on the input text stream: `flatMap` and `map`. The former applies the function `split` to each element of the input stream to split it into the lists of substrings using the separator semicolon and flatten the result into a stream of substrings, each of which is transformed into an integer value (or `None` if a string contains other symbols than alphanumeric digits). The resulting list of integer values is assigned to the variable `intList`. In line 5, the chain of monadic methods `map` and `reduce` transform each integer into the integer value one and apply the summation to determine the number of all input integer values. The method `collect` gathers all values from different nodes to tally them and store the result into the variable `total`. In line 6, the number of input negative integer values is computed by filtering out positive values and in line 7 the sum of all positive integers in the list is computed. Finally, in line 8 the average is calculated using the previously computed values.

Question 6: Is it possible to create an algorithm in Spark that would automatically determine when and where to apply the method `collect`?

This example is highly representative of the core functionality of Spark, where the main benefit of the Spark abstractions is to reduce the complexity that a programmer has to deal with when creating an application that manipulates distributed objects. In Spark, same RDDs can be reused in multiple iterations of the same algorithnm and these RDDs with the algorithmic operations are distributed automatically and efficiently across multiple computing nodes. Nevertheless, programmers view a single

program withour reasoning about these multiple nodes thus preserving location and migration transparencies. The bulk of the complexity is handled by the underlying platform called Mesos on top of which Spark is built.

13.4 The Architecture of Mesos

Mesos is a software platform that enables multiple cloud computing frameworks (e.g., Hadoop) to share computing resources granularly when running multiple data processing processes on different computing nodes while reducing unnecessary data replication among these resources and improving the performance of these jobs by achieving high data locality [73]. Mesos uses ZooKeeper[1], an open source service to allow processes that run on different nodes to coordinate operations via its hierarchically organized namespaces. Each process can atomically broadcast messages to other processes and receive them via a replicated database. Zookeeper has high performance and reliability.

The key idea of Mesos is to allow computing frameworks that run on top of Mesos to schedule and execute tasks using their own heuristics rather than to require these frameworks to delegate task scheduling and execution to Mesos. The main internal components of Mesos is the master module that controls and manipulates one or more slave components whose processes run on the corresponding computing nodes in the cloud. The master process has one or more standby master processes for fault tolerance, and these master processes cooperate using ZooKeeper. The master process communicates with specific framework schedulers to coordinate their tasks on the corresponding slave nodes that run framework executors.

The Mesos master process determines how many resources are available on all computing nodes that run its slaves. Once a framework scheduler starts (e.g., Hadoop or Spark) its scheduler registers with the Mesos master process, so that Mesos can offer resources to the framework tasks. However, it is up to each framework scheduler to decide which of the offered resources can be accepted and then frameworks send task descriptions to the Mesos master to run on the nodes using the accepted resources. Alternatively, a framework can reject the resource offers from the Mesos master, for example, because the offered resources may be insufficient to run a task or because the task may be cancelled as part of some assertion failure.

Question 7: How is Mesos scheduling algorithm different from YARN in Hadoop? How is it different from Kubernetes?

Consider a Mesos-based system with tens of thousands of computing nodes and tens of frameworks running dozen of tasks each. To achieve scalability, it is important to reduce the number of messages that the Mesos master exchanges with its framework schedulers. Particularly, reducing the number of messages associated with proposed and rejected resources have a positive impact on the scalability of Mesos. To achieve

[1] https://zookeeper.apache.org/

this reduction, framework use Mesos filter interfaces where frameworks pass their resource constraints to the Mesos master as lazily evaluated Boolean predicates. Doing so enables the Mesos master select only those resources that pass constrains checks thereby avoiding sending those resource offers that are known to be rejected by some frameworks in advance. Also, sent resource offers that pass constraint checks can still be rejected by the frameworks if certain runtime conditions warrant such rejections.

One main assumption behind the design of Mesos is that most tasks' durations are short. It means that Mesos master is designed to work as a fast switch allocating resources to tasks that run for a short period of time, deallocating the pre-assigned resources once tasks finish and reallocating these resources again to waiting tasks. It is easy to see that if the assumption is violated frequently, Mesos will lose its efficiency.

To address this problem, Mesos determines if many long tasks hogged resources, its master will inform the offending frameworks that it would kill its tasks after some grace period. At the same time, it is quite possible to have long-running tasks, so Mesos allows frameworks to inform the master that certain tasks will require guaranteed resource allocation without killing them.

Question 8: Compare the Mesos scheduling algorithm with Google Borg's and Omega's ones.

Finally, Mesos provides a strong level of isolation among resources using the concept of *chroot* or *chroot jailing* that became a part of Unix circa 1982. Recall that the operation *chroot* allows users to run a program with a root directory that can be freely chosen as any node in the file system, and the running program cannot access or manipulate files outside the designated directory tree, which is called a *chroot jail*. The chrooting mechanism is used in Mesos to prevent access to the same resources by running tasks.

13.5 Delayed Scheduling

A key benefit of Mesos is to allow the master to multiplex nodes to frameworks and their tasks fairly while maximizing the performance of executing these tasks. Fairness is achieved by terminating long-running processes to make resources available for tasks that have been waiting to run for some time. Of course, one can argue that a task that requires more time to run may be treated unfairly if it is terminated frequently and the computation is thus wasted. Waiting on the task longer to complete may result in better fairness overall. The performance is improved by assigning tasks to nodes that already have data that these tasks require thus improving data locality.

Delayed scheduling is based on the idea that when a scheduled-to-execute task may wait for a few particular nodes that has already data available for this task and the wait is generally short because most tasks finish quickly and the wait for one of these preferred nodes will be exponentially decreasing in the limit as time goes by [159]. To show how it happens, let us formalize the problem.

Suppose that a cloud computing cluster has M nodes with L task slots per nodes, where a slot is an abstraction for available resources to execute some task, and all tasks take approximately the same time, T to run. Let P_j designate the set of nodes that have data for the job, j. Thus, the ratio of preferred nodes for j is $p_j = \frac{|P_j|}{M}$ where $|P_j|$ is the cardinality of the set P_j. Also, p_j is the likelihood that the job, j has data on each slot that becomes free at some point of time. Suppose that these slots become free in some sequence and first available slots do not have the data for the job, j. Then, the job, j can wait up to D slots until the master decides that the job must take a slot that does not have the local data for this job. The main result that we will show next is that the non-locality decreases with D exponentially.

The likelihood that a task does not get a slot with local data is $1-p_j$ and since the availability of each slot is an independent event, we can multiply these likelihoods for D sequentially appearing slots with non-local data: $(1-p_j)^D$. Since each slot becomes available on average each $\frac{T}{S}$ seconds, the job, j will wait at most $\frac{TD}{S}$ seconds. Choosing the value of D becomes a critical choice.

Question 9: What is a ballpark good ratio of the long running vs short running jobs for the delay scheduling approach to be more effective and efficient?

Suppose that the goal is to achieve the overall job locality, $0 \leq \lambda \leq 1$, where the value zero means no task has local data and the value one means that all tasks are assigned to nodes with the local data. Suppose that there are N tasks that the job, j must run on the cluster with the replication data coefficient, $1 \leq \rho \leq M$, where the value one means no data is replicated across nodes and the value M means that the data has replicas on all nodes. The likelihood that some node does not have a replica to run the kth task is $(1-\frac{k}{M})^\rho$ and the formula $p_j = 1-(1-\frac{k}{M})^\rho$ for the task $1 \leq k \leq N$. The reader recognizes the famous compound interest inequality in the limit: $(1-\frac{r}{n})^{nt} \leq e^{rt}$ for $n \to \infty$. Applying this inequality we obtain the likelihood that the job, j starts a task with the local data by waiting D is $1-(1-p_j)^D = 1-(1-\frac{k}{M})^{D\rho} \geq 1-e^{-\frac{\rho Dk}{M}}$.

At this point, let us determine the average of the job locality, l given the waiting time, D:

$$l(D) = \frac{1}{N}\sum_{k=1}^{N} 1-e^{-\frac{\rho Dk}{M}} = 1-\frac{1}{N}\sum_{k=1}^{N} e^{-\frac{\rho Dk}{M}} \geq 1-\frac{1}{N}\sum_{k=1}^{\infty} e^{-\frac{\rho Dk}{M}} \geq 1-\frac{e^{-\frac{\rho D}{M}}}{N(1-e^{\frac{\rho D}{M}})}$$

. Now, we solve $L(D) \geq \lambda$ and obtain $D \geq -\frac{M}{\rho} ln(\frac{N(1-\lambda)}{1+N(1-\lambda)})$[2]. As as example, assuming that $\lambda = 0.95$ where only five percent of nodes do not have local data for the tasks and the job, j has $N = 20$ tasks and data is replicated on three nodes, $\rho = 3$, we obtain $D \geq 0.23M$. Assuming that there are eight slots per node, $L = 8$, we obtain D to be 2.8% of the average task execution time, i.e., $0.28T$. Thus, delayed scheduling algorithm is a key part of Mesos to balance fairness with waiting time for data locality to achieve high performance of executing tasks.

[2]This derivation is left out as a simple exercise

13.6 Spark SQL

Spark SQL (Structured Query Language) is the next step in the evolution of Spark functionality was to integrate its core functions with the relational data model processing [15]. A relation is defined as a table with columns called attributes and rows of data for each of the defined attributes. Tables may be related to one another via constraints on the data they hold. For example, the table EMPLOYEE is related to the table DEPARTMENTS, since each employee is assigned to work at some department. It means for each value for the attribute department in the table EMPLOYEE there must be a correponding value in the table DEPARTMENTS. Relational data model is widely used and many software applications have been created that retrieve and store data using databases that implement the relational data model.

SQL is a declarative language that is based on the relational algebra that in turn defines five basic operations: selection retrieves a subset of rows from a relation, projection deletes some attributes from the retrieved data, cartesian product combines data from two relations, set difference retrieves data that are present in one relation and absent in some other relation, and union retrieves data from all specified relations. Relations can also be intersected, joined, and renamed.

Executing complex relational operations is performance-sensitive on relations that contain big data. Consider the conditional join on two relations, $R \bowtie_c S \equiv \sigma_c(R \times S)$ where σ is the selection operator and c is some condition, e.g., R(departmentID) $=$ S(departmentID). However, this conditional join can be performed using a different sequence of relational operators: $R \bowtie_c S \equiv \sigma_c(R) \times \sigma_c(S)$. Suppose that each relation has $10,000$ rows and only 100 rows satisfy the condition. The former sequence of relational operators will produce an intermediate relation for $R \times S$ that contains $10,000,000$ rows before applying the selection operator, whereas the latter sequence of relational operators will not expand the initial relations to such a big intermediate one. The job of query optimization is performed by the underlying database engines, freeing SQL programmers from having to reason about the enormous complexity of the query optimization space.

Question 10: Discuss how referential transparency enables automatic optimizations of the data manipulation programs.

Relational database optimization engines are complex programs that take an SQL statement as the input and transform it into a sequence of low-level data access and manipulation functions to accomplish the goal that programmers encode in the SQL statements. Many SQL statements result in a very large number of combinations of the low-level functions to compute the result and selecting the optimal combination that minimizes the execution time is an undecidable problem in general.

Consider an example of an SQL statement that is shown in Figure 13.2. It retrieves the information about employees of some common gender who work in the same department. The statement is in fact a nested query that contains two SQL statements on the same table. Depending on the API used by an application, this query is sent to a relational database engine, which processes and optimizes the query and then returns

the resulting data to the application for further processing.

```
SELECT a.* FROM EMPLOYEES AS a INNER JOIN (
        SELECT DEPARTMENT_ID, COUNT(*) FROM EMPLOYEES
        WHERE GENDER = 1 GROUP BY DEPARTMENT_ID, DEPARTMENT_NAME
        AGGREGATE COUNT(NAME)) AS b
ON a.department_id = b.department_id
```

Figure 13.2: An SQL statement that retrieves information about employees.

Generally speaking, the application may look like the one shown in Figure 13.1 in which we can insert this SQL query as a string parameter to some API call that sends this query to the corresponding database for execution. However, the table `EMPLOYEES` can be very big and it is unclear how it can be partitioned into RDDs, and even if it does, say by distributing columns or groups or rows to different nodes, then how does Spark optimize the query to retrieve the data with good performance?

Spark SQL addresses this question – it is a library on top of Spark that presents abstractions to programmers who need to write SQL statements to retrieve data from relational datasets. These abstractions are realized in interfaces that can be accessed using a database open access software like *Open DataBase Connectivity (ODBC)* or with the Spark SQL's DataFrame API. An example of the Spark SQL's implementation of the SQL query from Figure 13.2 is shown in Figure 13.3. Some names slightly changed, it is easy to see how similar these queries are except that the SQL query is encoded using DataFrame API calls. Doing so enables Spark SQL to generate an execution plan to retrieve the resulting data and to perform optimizations of the generated plan to distribute tasks among nodes to improve the performance.

```
val employeeData = employees.as("a")
.join(employees.as("b"), $"a.deptID" === $"b.deptID")
.where(employees("gender") === 1).agg(count(name))
.groupBy(a("deptID"), a("name"))
```

Figure 13.3: A Spark SQL statement that retrieves information about employees.

The query optimization function within Spark SQL is performed by the Catalyst optimizer, an extensible component that is based on Scala programming language. At its core, the Catalyst optimizer contains a library for analyzing the DataFrame queries in their tree representations and generating multiple execution plans using custom-defined rules. Then these plans are evaluated and the winning execution plan is chosen.

Question 11: Is it possible to perform effective optimization of the Spark Dataframes if programmers write code like it is shown in Figure 13.3 where all string constants are replaced with variables whose values are computed at runtime?

Representing Spark SQL statements are trees is straightforward – the translation is similar to creating a syntax tree, where nodes represent expressions and literals and edges represent the nesting and connectedness of these expressions [6]. For example,

for the expression `agg(count(name))` the top root node of the tree represents the call to `agg` and the child of this root node is a node that represents a call to the function `count` whose child node in turn represents the literal "name". Constructing a parse tree is the first step in Catalyst optimizer.

Transforming a parse tree involves traversing its nodes and applying rules that replace nodes and branches into some other nodes and branches to reduce the execution time while maintaining the semantics of the query that is represented by the tree. Transformation rules are expressed in Spark SQL using Scala pattern matching, where transformations can be viewed as `if-then` rules with the antecendent describing the substructure of the tree and the consequent specifying the actions that should be performed on the instances of trees that matched the structure. For example, the following rule `case Square(Var(x)) =>Multiply(Var(x),Var(x))` tranforms an expensive function that computes the square of a variable into the multiplication of the variable value by itself, which is often cheaper. Spark SQL transformation rules are much more complicated in general and they can contain arbitrary Scala code.

After logical optimization where rules are applied to transform the parse tree and the physical planning phase that replaces DataFrame API calls with low-level operators, the resulting code is generated from these low-level operators. To do it, the parse tree must be traversed internally by the Spark SQL library. This task is accomplished using Scala quasiquotes, a language-based mechanism for translating programming expressions internally into parse trees and storing them in variables as well as unparsing trees into programming expressions [139]. For example, consider a scala variable, `val v = 1`. Using a quasiquote, we can obtain a tree `val tree = q"$v + $v"`, where `tree: Tree = 1.$plus(1))`. Interested readers can create an sbt-based Scala project, include the dependency `libraryDependencies += "org.scala-lang" % "scala-reflect" % scalaVersion.value` in the file `build.sbt` and use the module `scala.reflect.runtime.universe`. Once the tree is constructed, rules are applied to transform the tree into the syntactivally correct and semantically equivalent Scala program that Spark will execute on the destination nodes.

13.7 Summary

In this section, we studied Spark and its underlying mechanisms for executing big data processing applications in the cloud. Main abstractions in Spark enable programmers to concentrate on the data processing logic of their applications and avoid complicated and error-prone reasoning about data location, access, migration, and other kinds of data manipulations to improve the performance. The underlying platform called Mesos handles other frameworks besides Spark and it allows these frameworks to specify what resources Mesos should provide for executing their tasks. Moreover, the delayed scheduling technique helps Spark balance the fairness of executing diverse tasks and improving the overall performance of the system. Finally, Spark SQL shows how extensible Spark is for processing relational datasets and how it transplant relational database optimizations techniques using Scala language mechanisms.

Chapter 14

The CAP Theorem

Imagine yourself at your computer browsing and purchasing items in a web store. You select an item you like and put it in a shopping cart and start the checkout process. Unbeknownst to you, there are hundreds of shoppers who like this item and attempt to buy it. Yet, there is a limited quantity of this item in the company's warerhouse, since the space is limited and expensive and the company cannot predict with a high degree of precision the demand on all items. Nevertheless, all shoppers proceed with their purchases of this item and receive the receipt stating the delivery date within the next day or two.

However, within a day a few shoppers will receive an email informing them that their item is on backorder and the shipment will be delayed by a couple of days. Few of us get deeply frustrated with this experience – we shrug it off and we don't overly complain about a couple of days of delay. When we decompose this situation, it is clear that the web store allowed us to purchase an item that they could not guarantee to have in stock to satisfy all incoming orders. Instead of putting our shopping carts on hold and telling it that it may resolve a few hours to determine if the requested item is available, the web store optimizes our shopping experience by fibbing that the item was available in unlimited quantities and we could proceed with the purchase. Some web stores actually show a label that states that the item's quantity is limited, but many shoppers take it as a marketing ploy to boost the item's demand. One way or the other, the harsh reality is that we accept incorrect data from web stores in return for fast and pleasant experience when shopping for items.

This example illustrates an important problem that we will describe in this chapter and discuss solutions – how to balance the correctness of the results of the computation with the need to meet some performance characteristics of the application. When requests are sent to a remote object, it can queue these requests and process them, or it can respond with approximately correct results immediately and perform the computations in batches at a later time. Of course, ideally, we want our remote distributed objects to reply to requests with correct replies immediately, but as a sage once said, experience is what we get when we don't get what we want. Now get ready to get the experience of the trade-off between the correctness of the computation and the timing of the delivery of these correct results to clients in the presence of network failures.

14.1 Consistency, Availability and Partitioning

Recall that the term *availability* specifies the proportion of the operational time within specified constraints for the system w.r.t. some time interval. For example, a cloud SLA may specify that it stores and retrieves your data in blocks less than 1Gb within one second with three nines availability, i.e., the cloud may not be available w.r.t. the specified constraints approximately nine hours a year. The SLA may add that the *reliability* of the system is six nines, i.e., there is some small probability of 0.000001% a day of data corruption, i.e., assuming the maximum guaranteed data transfer of 1Gb a second or 86.4Tb a day, approximately 864Kb of this data may be corrupted. Additional measures like *mean time to failure (MTTF)* may specify that an average period of time within which a system functions without failures is five hours and components that fail will be managed through redundancy or error correction codes. These definitions reveal some interesting states of the system when it is available but unreliable, delivering wrong responses fast.

> **Question 1**: What types of applications are insensitive to the inherent unreliability of the cloud-based deployment?

The other important characteristic of the computation is how data integrity is enforced – computations must maintain and assure the accuracy and consistency of the data. A prevailing notion of creating software applications for many decades was to implement its functionality correctly to satisfy the functional requirements and then to worry about non-functional characteristics (e.g., performance and security) later. An illustration of a three-tier application architecture is shown in Figure 14.1 where clients from mobile devices on the left communicate with the back-end databases via the middle tier that contains the business logic implementation, i.e., program modules that implement some business requirements. Different I/O mechanisms are used in the process of data transfer and business logic execution. A notion of the "happy path" was created to illustrate the correct execution of the most important functional requirements where data storage/processing elements and program nodes form a path for a data flow from the clients to the back-end databases. Correct implementation of the happy path has been one of the important milestones in project execution.

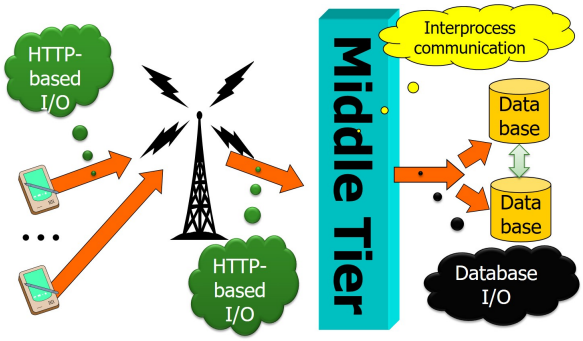

Figure 14.1: An illustration of a three-tier application.

However, with the advent of the Internet and the appearance of web-based busi-

nesses like Amazon, performance of web-based applications was taking over the correctness of the computations that they perform. Various studies repeatedly show that in the presence of competition among web-based businesses, an average customer/client is willing to wait for less than 15 seconds for a reply from the web business application. The switching cost is very small in most case, i.e., it take for a customer a few seconds to redirect her browser to the URL of a competing web store, whereas a trip to a physical store took much longer and the shopping experience with the switching cost was affected by the investment of time. Moreover, it is not just the fickleness of web store shoppers that makes the performance of the applications more important than the consistency of data – high-speed trading, search engines, social networks, content delivery applications, insurance fraud detection, credit card transaction management, revenue management, medical X-ray image analysis – the list of applications is growing where response time is more important than the correctness of the results.

Of course, in the days of yore of computing, applications were mostly monolithic and they exchanged a modicum of data by today's standards with a handful of backend databases. Failures arose from hardware and from logical errors made by programmers when they implemented requirements in applications. In distributed systems like the one that is shown in Figure 14.1 data loss or data corruption may happen at one or more points between the mobile devices and the databases. In fact, to reduce the load on the backend database, multiple instances of this database can be running, thus leading to situations where some data may not be updated and it takes some time to sync up (e.g., one database keeps orders whereas some other database stores customer transactions and the orders do not match the transactions for a given customer). Naturally, these situations can be avoided if transactional correctness is enforced.

14.2 Transactions and Strict Consistency Models

Dictionaries define a transaction as an interaction between people or some entities that results in the exchange of items. A business transaction is a set of operations to exchange goods, money, and services between people and enterprises [21]. Unless explicitly specified, all operations in a transaction must execute before the results of a transaction can be viewed. A program submits operations in a transaction to a database engine for execution and waits for its result. The database executes operations in this transaction and reports either success or failure to the program. Depending on the result, the program may issue the *commit* operation to make all changes made by the transaction operations permanent, or it may decide to *rollback* the transaction, in which case the state of the database is restored to the one that existed before the transaction was executed.

> **Question 2**: What happens when a virtual machine is shut down in which a transaction is executed?

The easiest way to think about transaction execution is in terms of strict consistency, where each operation is assigned a unique time slot and the result of the execution of

each operation is visible *instantaneously* to all entities in the system. Of course, no existing implementation of a computer can propagate the effects of writing to some storage location instantaneously, since various latencies are involved. In addition, the strict consistency model precludes concurrency, since only one operation on a given system can be executed in a given time slot. Despite the unrealistic constraints, this consistency model serves as a useful theoretical construct that shows the desirable level of consistency in transactions.

The sequential consistency model removes the instantaneous change propagation constraint and imposes a different rule whereby all writes to a storage location by all processors are ordered and their sequential order is observed to be the same by all observers (i.e., processors) in the system. In this context, it is important that each operation is *atomic*, that is, it is an indivisible entity that cannot be decomposed into some other, primitive operations. When a system is sequentially consistent, it can also be concurrent, as long as operations performed by different processors can be laid out in an ordered list, i.e., they are linearizable and their ordered list is equivalent to some sequential order that is thought to have existed by all entities in the system. That is, suppose that each transaction's operations are executed in some order, sequentially and without overlapping, and without any other transactions running in parallel and concurrently in the same system. Then, the same order and the outcome are preserved in the sequential consistency model when multiple transactions are executed concurrently in the same system.

Consistency models are embodied in the properties of *Atomicity, Consistency, Isolation, and Durability (ACID)* that are adopted in many popular databases. *Atomicity* guarantees that each transaction executes as a whole and its internal state cannot be revealed as part of the execution, only the final result. *Consistency* designates the correctness of the data, which is enforced by a set of constraints. For example, if an employee identifier is designated as a unique primary key in a dataset, then after executing a transaction there cannot be two records with the same employee identifier in this dataset. So the correctness of the data is ensured by the database or some transaction processing system w.r.t. some invariants and constraints are specified to maintain the consistency of the data. Using our previous example where the data is not reconciled in the replicated databases, we could say that the data becomes inconsistent w.r.t. the constraint that specifies the equality of the database replicas.

> **Question 3**: Does it mean for ACID-compliant applications that they execute all instructions of the same transaction in the same order?

The third property, isolation provides the sequential consistency model in the concurrent transaction processing context. Without isolation, race conditions are possible and they will make data inconsistent. Interestingly, changing the atomicity of a transaction, e.g., splitting it into two atomic transactions will result in exposing its internal state that may be changed by some other concurrently executing transaction leading to the loss of data consistency, however, in this case, it would be the fault of the software engineer who made the decision. For example, a decision to split a flight reservation into two or more transactions where each transaction reserves seats only for each leg

of the itinerary may lead to situations where there may not be any connecting flights available for a seat reserved for some leg. Since many concurrent transactions reserve seats for different passengers, by the time that one transaction reserves a seat, other transactions may take all seats available in other connecting flights resulting in wasted computing effort. However, doing so may be a part of the design, in some systems.

Isolation property is important when two transactions operate on the same data, e.g., two persons make withdrawal from the same bank account at two different locations – without isolation the race condition will lead to the incorrect balance value of the account. With isolations, locks are placed on data items within the database to ensure that the result is equal to the one obtained using the sequential consistent model.

Finally, the property durability specifies that the output of a transaction should be stored on some reliable storage and in the case that this storage experiences certain types of problems, the stored data should not be lost. There is some interesting interplay with consistency – the durability of the database may fail and some data may be lost, but the remaining data will be consistent w.r.t. the database constraints and invariants.

14.3 Two-Phase Commit Protocol

Suppose that a customer browses items in a web store, selects some items and places them in the shopping basket, and then decides to purchase them. As part of purchasing, the customer supplies the credit card and the shipping address and the web store calculates the shipping rate, all applicable taxes, allocates the requested type of packaging (e.g., a gift wrap), reserves a storage space for a parcel, and schedules the shipment. All these steps are performed by different web services that interact with multiple participants (e.g., United States Postal Service and local state taxation databases). Each of these steps is a part of the transaction and the transaction is committed if and only if each of these steps is committed by its respective service. To enable the ACID properties of a distributed transaction, the *Two-Phase Commit (2PC) protocol* is applied.

On the surface, a solution appears simple enough – let all services that participate in the transaction agree with one another that the transaction can be committed and then each service commits its steps. In the ideal world it would be the case, however, there is a problem with the order of $O(N^2)$ messages that N must exchange in order for each service to receive confirmations from the other services that they are ready to commit the transaction. Clearly, this protocol is not scalable, but it can be fixed by introducing a service that we call the *Coordinator (C)* and all services (S) communicate with C to decide on the commit leading to the order of $O(N)$ messages.

> **Question 4**: Does combining many smaller transactions in a large one in an application reduce the overhead of the 2PC?

Unfortunately, in the presence of failures this simple protocol does not suffice to ensure ACID properties: some nonempty subsets of the messages can be lost and S and C can crash and then restart at arbitrary times during committing a transaction. A key premise of the 2PC protocol is that every participant, S and C of the transaction

has a durable storage that it uses to write and read its sequences of transaction steps. Writing information about the transaction steps on the durable storage is not commiting a transaction, it is simply creating a log entry that describes the operations; this entry can be made permanent, i.e., committed or it can be aborted. These commands are issued by the program (e.g., a web store) that executes a transaction.

Doing so is accomplished by the 2PC protocol in two phases: `prepare` and `commit`. In the phase `prepare`, C sends a corresponding message to each S (e.g., *ready*) to get the transaction ready. Once an S stores its sequence of steps on its durable storage, it responds with a message that acknowledges that it is ready (e.g., *prepared*). When the C receives the message `prepared` from all Ses, it will mark the completion of the first phase of the protocol and it will initiate the second phase where it sends the message *commit* to all Ses that will proceed with committing the transaction steps and when completed Ses will send acknowledgement messages back to C. Once all acknowledgement messages are received from Ses, Ces will record that the transaction is committed and no further action is required.

In case when messages are lost or have some latency beyond an acceptable limit, C uses a timeout to send the message *abort* to Ses. If a message sent from C is lost, Ses may continue to wait, and once the timeout is reached, C will abort the transaction and restart it again. The same reasoning applies when a message is lost from some S to C. The chosen timeout values vary greatly and they may impact the overall performance of the application significantly.

> **Question 5**: Is it possible to design an algorithm that calculates the optimal value of the timeout for the 2PC protocol?

If one or more of the Ses crash, a few scenarios are possible. The crash can happen before the transaction is written into the local durable storage, after it is written, and if the crashed S restarts and how soon. Often a crash of an S means that C does not receive a response and the situation is handled with the timeout. If an S restarts soon and it wrote the transaction steps before the crash, it will respond with the message prepared to C and its crash will be masked. If a crashed S does not restart at all, then it means that the node on which this S is running has some serious damage and it may have to be replaced with a possible serious impact on the application's performance.

The crash of C is handled similarly. If it crashed in the first phase, it will simply repeat it after the restart. Since C also records its transaction state on the durable storage, it will obtain the information if all Ses responded in the second phase and it will proceed accordingly. The simplicity of 2PC and its robustness makes it a standard protocol to ensure ACID properties in most distributed systems.

Let us consider the complexity of the 2PC protocol in terms of the number of messages it exchanges, since each message exchange involves some latency and it consumes resources. For N servers, C sends N messages ready to which Ses respond with N messages prepared bringing the total to $2N$ messages. The second phase also involves $2N$ messages bringing the total to $4N$ messages. Writing transaction data into the durable storage can also be viewed as sending $2N+1$ messages to the filesystem managers bringing the total to $6N+1$ as the best case scenario without having failures

and without considering the overhead of memory and data manipulation. In practice, with complex distributed server topologies and sophisticated multistep transactions the overhead of the 2PC protocol is significant and often intolerable.

14.4 The CAP Theorem

The discussions on the balance between the availability and consistency started in the early 1980s, however, the issue rarely came up when creating software applications back then [41]. In the end of 1990s, creating large-scale distributed applications that communicated over the Internet became a reality. Dr. Brewer formulated a hypothesis in 2000 that specified a limit on distributed data-sharing systems, which can provide at most two out of three properties: **C**onsistency, **A**vailability, and **P**artitioning or CAP properties [31]. The reasoning on which the CAP hypothesis is based on the transition from the computation-based applications to distributed ones with multiple datacenters. With the introduction of mobile devices, the location of the data mattered more than before – latencies and the transactional overhead to preserve ACID properties made the system less available especially in the presence of network partitioning failures, which results in lost and corrupted messages.

> **Question 6**: Discuss an analogy between the CAP properties and three properties of economies: the cost, the availability and the quality of goods. How to ensure that all three properties are guaranteed, say in the health market? Analogously, how can one guarantee that it is possible to build a distributed application with the guaranteed CAP properties?

The CAP theorem was formulated and proved in two variants: for asynchronous and partially synchronous network models [61]. We state the theorems below and explain the proof in a way that shows why the statement of the theorem holds true when constructing a system is attempted with three properties holding.

Theorem 14.4.1 (CAP Theorem: asynchronous)**.** *An algorithm cannot be constructed to read and write a data object and its replica using an asynchronous network model where availability and atomic consistency are guaranteed in all fair executions in which messages can be lost.*

To understand this theorem and why Dr.Brewer's hypothesis should be formulated as a theorem in the first place let us specify precisely the terms used in this theorem. Let us start with the notion of the algorithm that is constructed – it is a sequence of steps that involve read and write operations on some data object and its replica object. We assume that this algorithm will always terminate, i.e., it cannot execute forever and it must produce a result or an error code. We state that the execution time is not unbounded, however, we do not specify the time limit on the long executions, however, we assume that such time limit exists. A more concrete example of this abstract assumption is that an algorithm cannot have an infinitely executing loop without producing any results in this loop.

Moreover, the operations that this algorithm executes in its steps are atomically consistent. Using the definitions from Section 14.2 we state that each operation in the algorithm is a transaction and the sequential consistency model applies where all operations are viewed to execute in some agreed sequence on a single node with a strict happens-before order between these operations. A violation of the order of the operations or observing the intermediate state of the step is impossible in this setting.

Finally, the notion of the fair execution is a way to state that everything that can occur during an execution of the algorithm is correct to assume as a valid option in each specific execution. One can enumerate all possible execution scenarios of the algorithm that include exceptions and wrong inputs and each of the enumerated executions can be viewed as valid possible executions. Since all communications between objects in a distributed system are performed by exchanging messages, it is valid to assume that each of the messages may be lost in each fair execution. Lost messages are modeled by erecting a wall or a partition between distributed objects that prevents these messages from passing, hence we use the term network partitioning.

Of course, there is a problem enforcing network partitioning and availability. What do we do when a message is lost – the algorithm cannot wait forever for a response. Therefore, to make this context consistent, we enforce that the algorithm must respond to lost messages with atomic consistency. One can think of producing a random or sloppily computed response by a message sending node if some predefined timeout to receive a response to this message expired.

The context for the proof of the CAP theorem for asynchronous networks is shown in Figure 14.2. A partition shown as a solid black vertical line divides the distributed system into two subsystems G_1 and G_2. The algorithm is implemented in the following way. Each partition has a database that stored a data object, V in one partition and its replica in the other partition. The initial value of this data object is v_0 in both partitions. The sevice, S writes a new value, v_1 in the partition G_1 and the service, P reads the value of the replica in partition G_2. Solid black arrows to and from the services designate write and read operation in the exactly defined sequence: first the write operation comes and then the read operation. The states of the objects are synchronized using messages that are shown with the gray block arrow that crosses the divide. The network partitioning is indicated by the fat red circle on the divide that results in the loss of some messages.

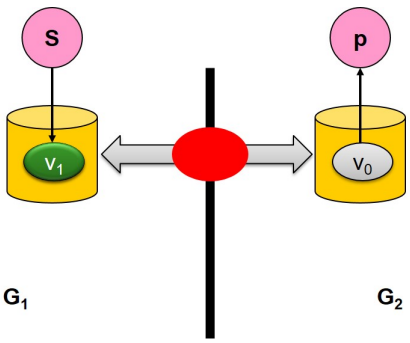

Figure 14.2: The context of the CAP theorem.

Suppose that there is an implementation of the algorithm where all three properties exist, i.e., the contradiction to the statement of the CAP theorem. It means the following. The initial state of the data object and its replica is the value v_0. Then the service S performs the transaction write that changes the value of the data object to v_1 with

the subsequent termination of the operation write. The assumption of the availability means that the operation write must terminate at some point of time and the service S must receive a message about the successful completion of the operation.

Next, the service P performs the transaction read on the data object replica. Since the assumption of the availability holds, P must receive the value of the data object replica. Since the data object and its replica can update their states using messages that cross the divide, if a message that carries the value of the transaction write, i.e., v_1 is lost, then the algorithm either waits forever until the data object replica is updated or it returns the value v_0 to P. The former violates the availability assumption and the latter violates the atomic consistency assumption. At the same time, assuming that all update messages that cross the divide are not lost violates the fair execution assumption with network partitioning. The proof is completed.

A variant of the distributed system for the CAP theorem is a partially synchronous distributed system where the notion of local time is introduced by assigning an abstract clock to each distributed node and assuming that there is no divergence between these clocks. That is, the time intervals passed in each node since some event occurs is the same. The purpose is not to define the global time, but to enable each node to determine how much time passed since an event occurred. The idea of using intervals in the proof is to determine when one transaction completes and the other begins. The format of the proof is similar to the asynchronous network proof within some time interval.

Question 7: Write down and explain the proof of the CAP theorem for a partially synchronous distributed system.

The CAP theorems directs the designers of the distributed systems to consider the following classes of these systems: consistent with partitions (CP), available with partitions (AP), and consistent with availability (CA). In CP distributed systems, the termination is not guaranteed, since the operations wait until all messages are delivered and all transactions execute. The 2PC protocol guarantees the ACID properties, however, the system may not be available due to waits and the high cost of transactions. Opposite to it, AP distributed systems relax the consistency requirement, so a possible result is that the clients will receive stale values that can be viewed as a result of the out-of-order execution of operations. Finally, CA systems are not distributed, since they assume that network partitioning events cannot happen.

14.5 Practical Considerations for CAP

As decades have passed since the introduction of the original CAP hypothesis, the question is how important is the CAP theorem in designing and developing large-scale distributed systems. In his assessment of the CAP theorem 10 years after the introduction of the initial CAP hypothesis, Dr.Brewer stated that one of the main uses of the CAP theorem is how to weaken the consistency in distributed applications to improve their availability – in CAP, A trumps C [30].

To understand why consistency is often sacrificied in favor of availability, let us look into the financial aspects of using distributed object technologies. As of the end of 2018, according to the Dun & Bradstreet Corporation, a company that provides business intelligence and analytics to businesses since 1841 and whose revenues approach $2Bil annually, 59% of Fortune 500 companies experience a over one and half hours of downtime per week. This means that an average Fortune 500 company with at least 10,000 employees pays an average of $56 per hour, including benefits ($40 per hour salary + $16 per hour in benefits), the labor part of downtime costs for an organization this size would be close to $900K weekly, translating into more than $46 million per year. On average, the businesses surveyed by the Dan & Bradstreet Corporation said they suffered 14 hours of technology downtime per year. Half of those surveyed businesses said these outages damage their reputation and 18% described the impact on their reputation as very damaging.

Next, let us check to see how much web-based businesses make in revenue per second that employ large-scale distributed electronic commerce platforms that produce these revenues. By taking the total revenue for year 2017 and dividing it by 31, 530,000 seconds in a year, we obtain the following numbers: Amazon earns $1,084 per second, Google earns $929 per second, eBay earns $290, Alibaba earns $176, Expedia earns $106, NetFlix earns $68, Facebook earns $63, and Skype earns $27 per second with $1 per second is equal to miserly $31,536,000 in annual revenues. These and many other companies use e-commerce platform that are composed of thousands of web services and tens of thousand of microservices, which implement functionality ranging from movie recommendations to business logistics. In these applications, (micro)services are exposed through well-defined interfaces and they are accessible over the Internet and they are hosted in an infrastructure that consists of millions of commodity servers spread across many data centers around the globe. It is easy to see how important availability is for the health of these businesses: with all other conditions fixed, higher availability means higher revenue!

Question 8: Assuming that we can quantify the willingness to pay for highly available services, design an algorithm that can automatically balance the CAP properties for a distributed application.

Of course, not every distributed application can relax consistency requirements. For example, an application that obtains the medical history of a person for emergency services may enforce strict consistency, especially when it comes to obtaining the information on life-threatening allergies before administering medications. High-speed trading where algorithms decide how to allocate funds across different investment products must base their decisions on completely consistent information about prior trades. Electric power grid applications must operate on timely data. These and a few other similar applications are consistency-critical; violating strict consistency may lead to the significant loss of money and resources and ultimately the human and animal lives. However, many e-commerce platforms are not mission critical – not obtaining the most recent (e.g., within a few minutes) movie ranking or post updates on a social network or the precise availability of an item in a web store will not lead to significant problems

for the absolute majority of the users. However, delays in presenting information to these users will lead them to switch to other, more available platform providers thus leading to significant losses of the revenue to less available commercial organizations and businesses. Hence, availability trumps consistency.

E-commerce application design is often guided to a certain degree by the 15-second rule that roughly states that if a website cannot interest the user in the first 15 seconds then many users will switch to the competition or leave the website altogether. A high bounce rate of users is indicative of a poorly constructed e-commerce platform. It is also widely believed that if a result of the query is not produced within some time interval, usually much less than 15 seconds, the users will bounce from the website. Therefore, even though the network can be very fast, the latency from enforcing strict consistency may lead to a high bounce rate and the downfall of a business. For example, the Amazons SLA rule limits the response time to 300ms for its web-based application [42]. According to Dr.Vogels who oversaw the design of the e-commerce platform at Amazon: "We do not want to relax consistency. Reality, however, forces us to."

Question 9: Is there a bottom limit on how fast services can be provided to users – obviously, it is the speed of light in the extreme case, but with respect to receiving a response from purchasing an item from a web store, does it make any difference for customers to receive a 100ms response vs 150ms response time?

As if designing distributed applications is not difficult in itself, nowadays software engineers must decide how to choose availability over consistency leading to significantly increased complexity of the design and implementation. Understanding latencies and how to replace ACID mechanisms with much faster but inconsistency accesses is important to ensure that high performance that leads to eventually applying data conflict resolution rules. Weak consistency models are born as a result of a trade-off between the complexity of e-commerce platforms and their high performance (i.e., availability).

Critiques of the CAP theorem are abound from the practicality point of view. Latencies and network partitioning are deeply related as delayed messages may be viewed as messages lost. On the other hand, with network transmission speed and quality improving every year, one may view some parts of the networks free from any partitioning events, hence, without significant effort applications running on these networks may be both highly available and consistent. Thus one of the critiques of the CAP theorem is that it offers binary dichotomy: either there is a network partitioning or not. In reality, network partitioning can be viewed in terms of an increased latency, where fault tolerant networks have multiple redundant pathways to ensure that messages are delivered within some latency threshold with the high probability. As a result, all three CAP properties can co-exist in the implementation with some weights.

When network partitions exist and messages are lost, two design strategies are commonly adopted to implement the CAP trade-off. First, conflicting operations are rescheduled to enable transactions to proceed without blocking access to data objects. Consider a social media platform as an example, where people mostly read posts and rarely leave comments on them. Write operations may be restricted in case of net-

work partitioning and applied later when network access is restored. Second, replicas or logs are used to record information about operations that cannot be performed due to network partitioning or that are avoided because applying them will lead to poorer performance. As a result, some users will obtain an inconsistent state of some objects during some intervals of time. The job of the software engineers is to determine how to resolve inconsistencies while keeping their applications highly available.

Question 10: Discuss the cost of resolving inconsistencies in a highly distributed software application with many replicas.

14.6 Summary

In this chapter, we introduced the context of the CAP theorem, showed its proof, and discussed the implications of the CAP theorem on the design and development of large distributed applications. We showed how the properties of consistency, availability and network partitioning are played against one another to obtain a distributed application that provides results of computations to users with high performance while allowing some inconsistency in the computations. The deeper issues are in selecting consistency models and the corresponding transactional support for distributed applications – depending on the mix of operations on data objects and the level of network partitioning, even high throughput networks will not help applications to increase their availability, since it would take beyond some allowed latency threshold to provide the desired level of consistency. We will use the framework introduced in the context of the CAP theorem to review relaxed consistency models and we will see how they are used in distributed applications.

Chapter 15

Soft State Replication

The CAP theorem serves as a theoretical underpinning for determining trade-offs between consistency and availability when designing large-scale distributed applications. A transaction that locks multiple resources until the 2PC protocol commits the transaction is often unrealistic in the setting where clients must receive responses to their requests within a few hundred milliseconds. There are multiple technical solution to implement weaker consistency models: selectively lock data objects to enable operations on these data objects to compete without waiting on one another, replicate data objects and assign certain operations to specific data objects while keeping the metainformation about these operations in a log (e.g., perform reads on one data object and writes on some other data object with a log keeping timestamps of these operations), or simply store all operations in log files and apply them in a large batch operation at some later point. One way or another, the clients receive fast and sometimes incorrect response values to their requests and the correctness will be achieved at a future time.

The value of strict consistency guaranteed by ACID transactions is often overestimated even in the domain of financial transactions. Suppose that two clients access the same bank account from two ATM machines that are located across the world from each other. One client credits the account whereas the other debits it. Applying the 2PC protocol to allow these transactions to leave the account's balance in the consistent state may result in the latency to one transaction at the expense of the other one. If it seems like no big deal for a couple of clients, think of a large organization that sells its product directly via its online store with thousands of purchase and return transactions performed each second. Creating multiple replicas of the organization's bank account and allowing these transactions to proceed in parallel without any locking will result in the inconsistent states of each replica, however, eventually, these replicas will be synchronized and their states may reach the consistent global state, say on the weekend when the store is closed. However, if the store is never closed, the synchronization of the replicas will continue in parallel to the business transaction, at a slower pace, perhaps, to avoid conflict and the entire application will fluctuate between various degrees of consistency, getting closer to the expected global state at some times and farther away from it at some other times, eventually converging to the expected global if all business transactions stop and the application reaches a quiescent state.

This is the essence of the notion of the *convergence property* [64] or better known as the *eventual consistency* [80, 149] that we will study in this chapter.

15.1 Consistency Models

Recall our discussion of the strict and sequential consistency models in Section 14.2. A formalized model for consistency models includes the set of processors that execute some algorithms that read from and write into locations in the shared memory [106]. Each processor keeps its history of the operations it has performed and the union of all local processor histories is the global history of the distributed application. To unify local histories, we consider the notions of the order and serializability. The former defines some rule where one operation in a history is considered to occur before some other operation in a different history. The latter is structuring the operations in different local histories in a sequence according to some order where there are no concurrently executing operations and all operations in the sequence come only from the chosen local histories [5]. Ideally, we want every operation read in the sequence to return the value that was written by a preceding operation write. Ideally, it should be the closest preceding write operation. An interesting aspect of serializability is that histories may be concurrent, however, the result of the execution is equivalent to the result of their operations executing non-concurrently, or strictly linear.

In the strict consistency model, all operations are synchronous and non-concurrent and they are positioned in a well-defined order. All reads return the value that is written by the most recent operation write and the client must wait for the execution of all operations until the response value is read and the communication latency is zero. In the more relaxed model of sequential consistency, one can specify a linear schedule where all operations are assigned some predefined position in a sequence such that all reads return the values of the most recent writes (i.e., a *coherent schedule*). In sequential consistency all processors execute their operations concurrently, and the result of this execution is equivalent to the execution of some coherent schedule of these operations.

> **Question 1**: For a set of five processors that execute read (R) and write (W) operations, where $R(x)v$ reads the value, v of the shared variable, x and $W(y)q$ writes the value, q into the shared variable, y, construct timelines with these operations that satisfy the strict or the sequential consistency models.

One of the weaker consistency models is called *causal* consistency where the causality between operations that occur in a distributed application are defined by messages that the objects that perform these operations send to each other [86]. That is, if some object, o_k sends a message to read the data from the object o_m and at some future time the object o_k sends a message to write some data into the object o_m, then the reading event precedes the writing event. However, if some object, o_p sends a message to write some data into the object o_m and there are no messages exchanged between the objects o_k and o_p, then it is uncertain what value the object o_k will receive in response to its message read – the one that was before or after the message write from the object

o_p was executed. Therefore, the causal consistency model is weaker, since it allows distributed objects to operate on stale data.

To illustrate causal consistency, suppose that the processor (or a client) P_1 writes values a and c into the same memory location, x, i.e., $W(x)(a),\ldots,W(x)(c)$. Independently, the processor (or a client) P_2 writes the value b into the same memory location, x, i.e., $\ldots,W(x)(b)$. The only order that we can infer from this description is the order between the operations write on P_1. There are no read operations in P_2 of the data written by P_1 and there is no other processor that can establish additional causal relations between these operations. Therefore, the following orders are valid as seen by some other processors: $R(x)(b),R(x)(a),R(x)(c)$, or the order $R(x)(a),R(x)(b),R(x)(c)$, or the order $R(x)(a),R(x)(c),R(x)(b)$ – the order or reads where values a and c appear remains the same as specified in P_1.

Now, let us assume that the processor P_2 executes the operation $R(x)(a)$ before its operation write. Doing so establishes a causal order between the operation write in P_1 and the operation read in P_2. Subsequently, the operation $W(x)(b)$ in P_2 is causally ordered after the operation read. Hence, the global history $R(x)(b),R(x)(a),R(x)(c)$ is not correct any more for the causal consistency model.

In the descending order by how weak consistency models are, the next one is called *First In First Out (FIFO)* or *Parallel RAM (PRAM)* consistency, also known as the program order or object consistency model [90,116]. In this model, the order in which each process performs operations read and write is preserved, however, nothing is guaranteed about the order in which mixed operations from different processes are performed. That is, the local histories are preserved for each process and the global history is not defined. Applying the FIFO consistency model to the ATM example, each client will see the exact order in which her debit and credit operations were performed on the distributed account object, however, the desired order is not guaranteed in which these operations are unified. Suppose that one client withdraws $100 and deposits $1000 from the account that has $100 while the other client withdraws $500 and deposits $100. If the operations performs in the described sequence, the account balance never drops below zero, however, it is possible that after the first client withdraws $100 and the balance drops to zero the other client withdraws $500 making the account delinquent. Thus, the FIFO consistency can be viewed as a further relaxation of the ACID property where transactions are not atomic and their internal states are exposed.

Question 2: For a set of five processors that execute read (R) and write (W) operations, where $R(x)v$ reads the value, v of the shared variable, x and $W(y)q$ writes the value, q into the shared variable, y, construct timelines with these operations that satisfy either the casual or the FIFO consistency models, but not both.

Even weaker is the *cache* consistency model in which every read operation on a shared memory location or a shared data object is guaranteed to obtain the most recently written value by some other operation. In this model, out-of-order executions are possible, since operations are concurrent, they are not synchronized and even the internal process operation order is not preserved [59, 102]. Think of a memory consistency model as a contract between the low-level hardware layer and the higher-level

software layer that contains operations that are executed by the underlying hardware. The hardware may have additional low-latency stores that we call caches (e.g., L1-L4 caches) that can store replicas of the values whose main store is in RAM. Suppose that an operation updated the value stored in the cache, but the corresponding location in the RAM is not updated instantly, there will be some time lag before it happens. That is, the location in the RAM stores the most recent value that is updated by some other operation, not by the operation that updated the value in the cache. In the meantime, the next read operation may read the value from the RAM and not the cache. As a result, the consistency is significantly weakened when compared to other models like sequential, causal, and even FIFO consistency models. Of course, if cache coherence is added to the cache consistency model, then all cache updates will be propagated to their original locations before processes read their values.Of course, combinations of these consistency models are possible and their implementations differ in difficulties when it comes to enforcing the constraints of these models on data replicas.

15.2 Data Object Replication Models

Replicating a resource or a process means to make more than one of its copy or replica available for operations to ensure redundancy in case some failures disable some of its replicas. Since replicas can be located on different types of storages, some of them may be more expensive and much faster than the others. Therefore, replicating objects enables software engineers to take the performance dimension into consideration besides fault tolerance – some operations may be scheduled on replicas positioned in L4 caches or the RAM and they will be executed much faster compared to replicas located on some network SSD storages with much higher latencies. Selecting a replication model and a proper consistency model affects the performance of a distributed application in a significant way.

There are two fundamental replication models: active and passive. A replication model is essentially an extension of the RPC client/server model where each server's primary data object has one or more replicas. An operation is completed on the primary data object, and it is repeated on all of its backup replicas. In the active replication model the operation is executed via some kind of broadcast on all replicas that update the data or respond to the read request. Opposite to it, the passive replication model has a single data object as the primary server that receives requests from clients and responds to them and it is also responsible to update its replicas, which may be executed at some later periods of time. These updates may vary from the strict or the *pessimistic* version where the response to clients is sent only if all backup replicas are updated to the latest values, to the *eager* or the *optimistic* version where the response to the clients' request is sent before updating the replicas. *Lazy* updates are also possible only if the data is requested from the replica and then its state is updated on demand.

Question 3: Explain designs of distributed applications where either the active or the passive replication model has the advantage over its counterpart.

In a seminal paper published in 1983 on designing a clearinghouse as a decentralized and replicated system, Derek Oppen and Yogen Dalal introduced the design of a clearinghouse for locating distributed objects [112]. The information about the objects was not supposed to be always correct – multiple inconsistent replicas of the data objects were expected to co-exist for some period of time until they would be reconciled, *eventually*. Importantly, a client-centric view is added as a perspective to the weak consistency model – in what ways clients perceive the violations of the strict consistency model? We will answer this question shortly in this section.

15.3 Software Design of Highly Available Applications

Dr.Butler Lampson, a Turing-award winner and one of the inventors of laser printers, two-phase commit protocol (2PC), the first What You See Is What You Get (WYSIWYG) text formatting program, and the Ethernet, published an influential paper on important properties of the design of software systems [87]. In this paper that has become a system design guide for generations of software engineers, Dr. Lampson introduced a framework that answers two main questions: what does a specific recommendation do (i.e., provides functionality, improves the performance, or makes the application fault-tolerant) and where in the application it should be implemented. His guidelines will help us understand the design of cloud-based applications.

One of the design guidelines is formulated as "leave it to the client." The idea is to allow the client to design and execute tasks they need to perform by using powerful and simple interfaces of objects that provide basic services. We do not mean the client only as a customer who purchases items from a web store, but rather in the RPC context where a client is a program that obtains services from the remote distributed object. The idea is that each distributed object must expose a simple interface with clear and concise semantics that solves one specific problem instead of a complex set of interconnected interfaces doing many different and unrelated tasks. Leaving it to the client means that a client of such distributed objects decides how to RPC them and process the resulting data. Conversely, distributed objects assume little knowledge of their clients.

An example is the design of monitors in the OS kernels that offer its clients a way to synchronize access to shared resources. Recall that monitors are very fast – it takes less than 20 nanoseconds to (un)lock a monitor. One of the reasons that monitors are so fast is that they perform one operation – they set and retrieve the lock state and nothing else. One can imagine many more functions for the monitor, for example, to detect if there is a waiting loop among applications using monitors to prevent deadlocks. However, adding this functionality will likely result in higher latency and heavier memory usage when compared to the delegation of the extra-tasks to clients like deadlock detection.

With the client/server model of the RPC, end-to-end execution trace of a distributed application starts with some "edge" client like a customer who uses a web browser to access some web store. On the "edge" of the other side there is a remote distributed object that stores and retrieves data items using some durable storage. An end-to-end or edge-to-edge execution trace of the main functionality is called a *happy path* if it includes sequences of activities that are executed if everything goes as expected without exceptions [28]. For a webstore, a happy path may be formulated as a sequence of the

following activities: find an item, put it in the shopping cart, select the checkout, enter the payment and shipping information and apply all relevant coupons, then push the button labeled `Purchase`. The result of this happy path is an email arriving within an hour to the customer's mailbox stating that the item is being shipped to the provided address and providing the breakdown of the charges.

This simple and straightforward happy path hides many technical issues, since it may be executed by hundreds of thousands clients per second. Consider the following services that must be used in the happy path transaction: wearhouse checks, delivery date confirmation, legal and export restrictions, communications with external vendors, credit card charge processing, shipping rate check, calculation of all applicable taxes, and determining the type of packaging, to name but a few. Some transactions may involve RPCs with hundreds of remote objects. Yet, the SLA guarantees that the customer will receive her response that confirms the transaction or shows why it failed in less than, say, 300ms. Clearly, a straightforward solution of connecting each customer directly to the backend database and applying the 2PC protocol will result in a serious performance penalty that will violate the SLA.

> **Question 4**: Does it make sense to apply the 2PC protocol only to a part of the happy path in a distributed application?

A key idea in the design of high availability distributed applications is to avoid synchronization when multiple clients access shared remote objects. Doing so may result in nasty race conditions and even crashes of some services, but there is a value in the guidance to separate the normal and the worst cases. If the crashes or some other unpleasant effects of applying solutions to increase the application's availability and speed up its execution happen infrequently without significant impacts on the quality of the results or user experience, then these solutions become a part of the toolkit for designing distributed applications.

Avoiding synchronization can be achieved using a variety of approaches. One is replication of distributed objects where concurrent requests are scheduled to different replicas, so that there is no need to use monitors to synchronize the access to these shared objects. As a result of this design decision, the application's state at a given time may not be equal to the *hard state* of the same application if the ACID properties are applied strictly. This *soft state* consists of different kinds of reconstructible data from partial transaction sequences and this soft state can be discarded and reconstructed later. In short, the hard state is the ideal state of the application when strict or sequential consistency is applied and the soft state is any other state of the application where the hard state is not enforced as a result of the trade-off between consistency and availability of the application.

Key technical solutions in the design of the highly available distributed applications are *caches* and *hints*, which short-circuit access to shared resources. The client/server architecture of distributed applications is organized in tiers and one of the most popular canonical architectures is a three-tier client/server architecture. Clients interact with objects in the first-tier that present interfaces for inputting data and displaying results. The second-tier or the middle-tier contains so-called business logic objects

whose methods implement various algorithms to process data to deliver the expected functionality to clients. Finally, the third-tier or the back end contains data objects that interact with a durable storage. Caches and hints are located in the first and the second tier to prevent, whenever possible expensive operations on the objects in the back end.

Caches short-circuit repetitive expensive computations by storing in memory the result of the previous computation in the first/second tier and delivering it to the client instead of repeating this computation. Using caches is rooted in the *principle of locality* that states that the same values or related storage locations are frequently accessed by the same application, depending on the memory access pattern [87]. By storing the relation $(f,i) \rightarrow f(i)$ as a memory-based map with f and i as keys, the previously computed value of $f(i)$ can be retrieved using a fast memory lookup. Of course, a cache may be limited in size and storing a smaller number of pre-computed values may result in cache misses, which incur some additional computation penalty and recomputing f for these missed entries at runtime may be quite expensive and performance will worsen correspondingly.

Question 5: Given a legacy application, explain how would you approach its redesign to use caches.

Unlike caching, a hint is the saved result of some computation, which may be wrong and it is not always stored as a map. Consider a situation when shipping charge is computed for an item ordered from a web store. Instead of contacting a shipping service with the zip code to which the item is going to be shipped, it is possible to look up the results of the other computations that is run in parallel to this one and select the shipping charge for the one whose zip code is the closest to the given location. Thus, the hint short-circuits an expensive computation and it returns an incorrect result, however, the eventually corrected shipping charge will not differ significantly from the one that was substituted from the closest zip code order. Even though a hint may be wrong, eventually checking its correctness and replacing the value is convenient and it improves the performance of the application. With caches, hints make distributed applications run faster. Deciding when to use caches, hints, and replicas is a design decision. Therefore, replicating a service without full synchronization forces designers to use hints and caches, and stale data will be returned to clients.

15.4 Eventual Consistency. BASE.

Since 1969, various engineering advancements in computer and electrical engineering are published as Request for Comments (RFC), a type of publication which is supported by the Internet Society. In 1975, RFC 677 was published by Johnson and Thomas on the maintenance of duplicate databases[1]. They stated the following: “Because of the inherent delay in communications between DBMPs, it is impossible to guarantee that the data bases are identical at all times. Rather, our goal is to guarantee that the copies

[1] `https://tools.ietf.org/html/rfc677`

are "consistent" with each other. By this we mean that given a cessation of update activity to any entry, and enough time for each DBMP to communicate with all other DBMPs, then the state of that entry (its existence and value) will be identical in all copies of the database" [79]. We adopt this statement with small modifications as the definition of eventual consistency. An application is eventually consistent if it makes a transition to a soft state and then it reaches its hard state in which all replicas are identical at some time in the future upon the cessation of all update activities.

Question 6: Is it possible to guarantee for an eventually consistent application that the full consistency will be reached within some time interval? How?

A major implementation of eventual consistency was described by eBay Corporation as their development methodology called *Basically Available replicated Soft state with Eventual consistency (BASE)*, in which "services that run in a single data center on a reliable network are deliberately engineered to use potentially stale or incorrect data, rejecting synchronization in favor of faster response, but running the risk of inconsistencies" [125]. Let us use the context of the CAP theorem that is shown in Figure 14.2 to clarify how BASE works. Suppose that both services *s* and *p* read and write data in their respective replicas in the presence on the network partition where the network is split into the partitions G_1 and G_2. With the strict consistency model, these replicas are locked for the duration of each transaction and when the locks are released, the application is in the consistent state. However, with eventual consistent model, we assume that the data replicas are convergent, i.e., there is a synchronization algorithm that applies operations performed by the services *s* and *p* in some sequence to bring the data replicas to the same final state in the absence of updates. In fact, the synchronization algorithm may be running concurrently with the operations of the services *s* and *p* and the network partition may appear and disappear at arbitrary periods of time. The question is what values and operations on the respective replicas the services *s* and *p* observe before the application reaches the quiescent state?

Question 7: Can a VM migration algorithm be formulated in terms of synchronizing distributed objects?

Let us organize the operations performed by both services in a sequence – we assume that the ideal application will implement sequential consistency in the absence of network partitions. Ideally, both services will "see" the entire sequence when the last operation is performed and they also see the same *prefix* of the sequence after each subsequent operation starting with the first one, where the prefix is the subset sequence of the operations. In this context, every service will read the data that is updated by the last write operation no matter what service performed this update. However, until the full consistent state will be converged to, the services will see different prefixes. A portion of the *consistent prefix* that all services see is one characterization of the degree of eventual consistency, where the state diverges after some operation in the consistent

prefix. A consistent prefix is a function of the *bounded staleness* constraint that dictates a certain time limit after which all operations become fully consistent. Services see different prefixes during the bounded staleness time interval that can be combined, generally into four categories.

Read your own writes means if the service performed the operation write before it performed the operation read, then the operation read will return the value written by that operation write. The service will never obtain the value written by out-of-sequence operation write, even though the service may obtain the value written by some other service that executes concurrently. This is an eventual consistency guarantee that is implemented in the Facebook Corporation's cloud.

Writes follow reads is an application of the causal consistency that specifies that if the operation write follows some previous operation read on the same data object by the same service, then this operation write will be performed on the same or the more recent value of the data object that was obtained by the previous operation read. An application of this guarantee is manifested in the ability to post a reply to a message on some social media that will be shown after the posted message, not before it.

Monotonic reads guarantees that once a value is obtained by some operation read will be also obtained as a part of the prefix by all services. Essentially, if there is the operation write that updated the value of a data object by the service *s* that will see this value by the following operation read, then this operation write and its effect will also be seen by some other service *p*. Violation of the guarantee of the read monotonicity would be a situation where services can obtain different values of previous operation write as time goes by.

Monotonic writes guarantees the order of writes on a data object by the same service. It is also most difficult to implement, since it involves a nontrivial amount of coordination among services to converge to the monotonically written state.

A recipe for implementation of BASE at eBay Corporation involves the following steps [125]. Given a transaction whose logical borders are delineated with the keywords Begin/Commit, we break this transaction by dividing it into operation steps that can be done in parallel. Each operation step is mapped onto a message that can be delivered to a server in the cloud to perform this operation. Message delivery is performed using queuing mechanisms - the idea of asynchronous message delivery is employed to send a message to the destination server and it will be eventually delivered and the operation will be executed. The broken transaction is overseen by some distributed object, a leader that runs the transaction and it stores these messages in a *Message Queuing Middleware (MQM)* system.

Consider a transaction in SQL that is shown in Figure 15.1. The start of the transaction is delineated by the keyword `Begin` in line 1 and it is committed in line 5. The statement `INSERT` in line 2 creates an entry in the table Order that records the sale of an item for a given amount of money between a seller and a buyer. The statement `UPDATE` in line 3 increases the balance of the corresponding vendor who sold this

```
1 Begin
2 INSERT INTO Order (Id, Seller, Buyer, Amount, Item)
3 UPDATE Vendor SET Balance = (Balance+$Amount) WHERE Id = Seller
4 UPDATE Client SET Balance = (Balance-$Amount) WHERE Id = Buyer
5 Commit
```

Figure 15.1: A sales ACID transaction that updates data in three database tables.

```
1 Begin
2 INSERT INTO Order (Id, Seller, Buyer, Amount, Item)
3 Queue Message("UPDATE Vendor", Balance+$Amount, Seller)
4 Queue Message("UPDATE Client", Balance-$Amount, Buyer)
5 Commit
```

Figure 15.2: A transformed sales transaction that sends messages to update data.

item by the given amount and the corresponding statement update in line 4 subtract this amount from the balance of the client who is the buyer. These three tables are represented by distributed data objects that are located on different server nodes. Performing this transaction with ACID guarantees will lock these tables for the duration of the transaction.

Question 8: Explain why is it important semantically to organize these SQL statements in an atomic transaction.

The first step of converting this transaction into a BASE update is to break it into two or three transactions with one SQL statement in each. Besides exposing the internal state to many other transactions, we still have problem with locking the distributed objects for the duration of their updates thereby preventing other transactions from reading/writing these objects. What we need is to inject some operation that allows us to convert an SQL statement into a message that will eventually be delivered and executed on some replicas of the distributed objects with some guarantees of the eventual consistency.

Consider a transformed transaction that is shown in Figure 15.2. Updates are converted into messages that take the necessary parameter values and these messages are sent and queued by some MQM. Of course, the state of the application become soft and to converge to the hard state the objects will have to be synchronized using some algorithm, for example, the one that is shown in Figure 15.3.

The idea of this synchronization algorithm is to update the tables Vendor and Client at some later time to make them consistent with the table Order. This is a possible synchronization algorithm that may not be applicable to many applications based on their needs to converge to the consistent data depending on the requirements. However, it illustrates how eventual consistency can be implemented in a generic way using MQM.

```
1  For each Message in the Queue Do
2   Peek Message
3    If there exists a transaction in Order with Message.Seller
4     Find Message in Queue with the corresponding Buyer
5      Begin
6          UPDATE Vendor with the corresponding parameters
7          UPDATE Client with the corresponding parameters
8      Commit
9      If Commit is Success, remove processed Messages
10 End For
```

Figure 15.3: Pseudocode shows possible synchronization.

15.5 Message-Queuing Middleware

MQM is a collective term for libraries and frameworks that provide distributed objects for queuing messages from producers and delivering them to consumer objects with certain guarantees, e.g., FIFO delivery possibly within some time interval, so that producers and consumers are loosely coupled. Message-queue-based interactions between producers and consumers are formalized in an open standard application layer protocol for MQM called *Advanced Message Queuing Protocol (AMQP)* [111, 156]. To enable fast and efficient message exchange in the enterprise-level distributed applications, AMQP is designed as a binary, TCP/IP protocol for flow-managed communication with various message-delivery guarantees and various security features. It is publically available[2].

The need for a common open-source protocol stems from the interoperability requirement to enable producers and consumers to exchange messages when the underlying vendor implementation of the MQM changes. That is, multiple MQM implementation may co-exist and messages can be send from a producer/consumer that uses one vendor's MQM to some other consumer/producer that uses a different vendor's MQM. To achieve this interoperability, it is not enough to adhere to some high-level API specification; messages must be presented in a binary format that is understood by vendors' MQM and the operations on these messages must be reduced to a common set that includes the following groups: operations for opening and closing connections, beginning and ending sessions, attaching and detaching links to connections, transferring messages, controlling the flow, and obtaining the status of the system.

The AMQP specification is composed of the following parts: types for interchangeable data representation, a model of the transport mechanism, message structures, transaction support, and security. These parts are essential components of any MQM. An application constructs a message as a sequence/set of values, and these values are defined by types that must be represented as sequence of bits when transmitted to the destinations. An example of an XML-based type definition for 32-bit floating point numbers is given in Figure 15.4, where the name of the type and its definition as a primitive (as opposed to composed types) type are given in line 1, its encoding name, the unique code, its category (i.e., fixed versus a variable length type) and width in

[2] `https://www.amqp.org`

```
1 <type name="float" class="primitive">
2   <encoding name="ieee-754" code="0x72" category="fixed" width="4"
3         label="IEEE 754-2008 binary32"/>
4 </type>
```

Figure 15.4: A transformed sales transaction that sends messages to update data.

bytes as well as a readable label are given in line 2 and line 3.

Using these easily extensible type definitions, an implementation of MQM can be extended to various platforms to transmit richly typed messages via AMQP transport network that consists of nodes and unidirectional links, where the latter establishes a stateful one-way communication channel that keeps track of the messages and the former are named distributed objects that store and transmit messages. A part of the node object that handles interactions with the link is called *terminus* and it can be either a source or a target of the *frame*, which is a transmitted message that encapsulates a unit of work. Frames are organized in sessions that establish logical grouping of frames between nodes. Frames may be transmitted (a)synchrononously and given certain time to live, i.e., if not delivered within this time limit, the frames will be dropped. Different levels of flow control enables applications to ensure measured issuance of messages depending on the capacity of the targets and the network bandwidth.

Question 9: Describe how a sequential consistency model can be implemented using MQM and what role the frames play in this implementation.

Transactional support is a major element in MQM – messages can be grouped in a transaction unit, where the actions of these messages are accomplished as an atomic work unit on commit or all of them are discarded on rollback. In the example that is shown in Figure 15.3 the messages update will be sent as part of the transaction; only after the updates are committed, the acknowledgements for the delivery of all transaction messages will be sent to the source node.

There are many MQM implementations of AMQP, both open-source and commercial enterprise-strength. Some of them offer proprietary functionalities to improve the scalability of the systems beyond the specification of AMQP. It is advisable to check Apache Foundation to determine what MQM projects are currently popular play around with their libraries and API calls to determine their extended capabilities.

15.6 Protocols For Convergent States

In order to synchronize the states of distributed objects and their replicas they must engage some protocol for exchanging their states. A simple and straightforward solution is to introduce some kind of distributed coordinator object, a central object to which all other distributed objects and their replicas send their states. Once received, the coordinator will compare different attributes of the states (e.g., update timestamps) and send messages back a la the 2PC protocol to update the states of the distributed objects and their replicas. All in all, it is an application of the 2PC protocol to committing states.

Of course, one big problem with this approach is that it is not scalable and having the central coordinator object introduces a serious bottleneck (e.g., an object that results in a serious performance degradation of the entire application) and a single failure point. Essentially, in a continuously running reactive distributed application applying this variant of the 2PC protocol will lead to the loss of availability and serious performance degradations. Thus, other protocols are needed that synchronize objects with their replicas with little overhead.

Question 10: Discuss the pros and cons of using the 2PC protocol only for a subset of the functionality of some distributed application to keep the states of some distributed objects in sync.

We discuss a few synchronization protocols as ideas in this section. Multiple papers and books are written on this topic and we will not repeat various algorithms here. Our goal is to give a general overview of what is accomplished in this area, so that the readers understand the cost of synchronizing distributed objects with their replicas.

One straightforward protocol is to perform reads locally without informing other objects, however, in response to an update, writes must be sent to all replicas to update their states. It is possible to group all replicas of a distributed object and multicast update messages to the entire group. Of course, the issue is the order in which the writes must be applied, since in a truly asynchronous application there is no global time to timestamp messages to ensure their order. To solve this problem, Lamport's logical clock is used where the relation happened-before is used to determine the order of messages [86]. In Lamport's system, a logical clock, $C : \mathbb{E} \rightarrow \mathbb{Z}$ is a function that maps some event $\in \mathbb{E}$ to some integer. One can visualize the execution time of each object with the logical clock where the execution timeline is represented by a line for each other and events are dots on these lines. Each initial event on each line is assigned some starting integer. If there are no messages sent from one object to another, then events are ordered only within the object timeline. That is, if a and b are events in some object j timeline, and a happens before b, then $C_j(a) < C_j(b)$. However, if the event a is the sending of a message by some object, j and b is the event of the receipt of that message by object, k, then $C_j(a) < C_k(b)$. The events that happen on the same timeline are assigned incremented integer timestamps starting with the initial one. However, if the event is a reception of a message from some other object, then its timestamp is the $\mathtt{max}(C_j(a), C_k(p)) + 1$ where the even p directly preceeds the event b. Using Lamport's logical clock the events can be totally ordered when synchronizing replicas.

Question 11: Discuss the overhead of keeping and comparing many versions of distributed objects. How can you reduce this overhead?

Another solution is to combine the idea of the logical clock with a centralized sequencer object. Each update is sent to the sequencer object that assigns a unique sequence integer to the update and forwards the update message to all replicas, which apply this update in the order of the sequence numbers assigned to them. Logical

clock-based algorithms are used in the number of cloud computing systems as we will learn later in the book.

A class of epidemic-inspired message exchanges can be broken into three categories: direct, anti-entropy, and rumor-mongering [43]. The problem with sending many messages between distributed objects and their replicas is that it generates dense network traffic when the number of object replicas increases, since it takes significant time to propagate update to all nodes. Instead, the idea of epidemic propagation is based on the limited number of messages that are sent from the updated object to its replicas and the other objects that need to be informed about the update.

In epidemic-inspired algorithms, distributed objects are viewed from the *SIR* perspective, where **S**usceptible are the nodes that have not received an update message, **I**nfective are the nodes that received an update and are ready to send it to other susceptible nodes, and **R**emoved are the nodes that received an update but will no longer send it to the other nodes. In the direct mail epidemic algorithm, each infected object will send an update message to all other susceptible object on its contact list. In the anti-entropy algorithm, every object regularly chooses some other susceptible object at random to send an update. Finally, with rumor mongering algorithms, an update message is shared while it is considered hot and timely. The more objects receive it, the less it spreads. Benefits of epidemic-inspired algorithms include high availability, fault-tolerance, tunable propagation rate, and the guaranteed convergence to the strict state by all replicas.

15.7 Case Study: Astrolabe

Astrolabe is a robust and scalable approach and a platform for distributed system monitoring, management, and data mining whose co-author is Dr.Werner Vogels, the Chief Technology Officer (CTO) at Amazon Corporation. In this section we will summarize salient technical aspects of Astrolabe and interested readers are directed to the original paper [151].

In Astrolabe, scalability and robustness are the key properties of distributed applications that are addressed in the design of the approach. Distributed computing entities are organized in hierarchical zones, where a zone is either a zone or a computing server. That is, an intermediate node in the hierarchy is a zone and a leaf node is a computing entity, a physical server or a distributed object. These leaf nodes contain *virtual zones* that have writable key/value storages that are populated with MIBs local to the node.

Hierarchical zones is a technique to achieve scalability by restricting communications within the hierarchy branches instead of allowing computing entities to communicate freely with one another in a large datacenter. Moreover, to propagate state updates, information from computing entities is summarized and aggregated into small data sets called *Management Information Blocks (MIBs)*, so that these MIBs are propagated within and between zones to synchronize states. Astrolabe is eventually consistent, where the update is dictated by a rule that states that if a MIB depends on some value that is updated, then the aggregated state eventually propagates through the system. Since different entities and zones may contain different types of data, Astrolabe allows administrators to add new aggregation functions to compute and update the aggregated

states, making the system flexible and scalable.

> **Question 12**: Give an example of an aggregated function for synchornizing the state of some distributed objects in a web store application.

Each computing entity must run an Astrolabe agent, which is a program that enables these computing entities to exchange information within the zone using an epidemic gossip protocol. Thus, there is no central distributed object that must collect the information and update zones and computing entities – with no single point of failure like that, Astrolabe is robust.

In Astrolabe, each zone randomly chooses some of its sibling zones using some predefined time period and these zones exchange MIBs choosing the one with the most recent timestamp. The first issue is the propagation of the information about zone membership – in a dynamic open environment computing entities join and leave at arbitrary time interval and frequently meaning that a significant amount of information about the membership traffic can be generated. The hierarchical organization allows Astrolabe the prevent this information from propagation systemwide, keeping it localized within particular zone hierarchies.

Astrolabe uses the following rule for eventual consistency: "given an aggregate attribute X that depends on some other attribute Y, Astrolabe guarantees with probability 1 that when an update **u** is made to Y, either **u** itself, or an update to Y made after **u**, is eventually reflected in X." Interestingly, the guarantees are not given on the time bounds for the eventually consistent update propagations, but the word "eventual" specifies that it will happen in the future. To see the results of the experimental evaluation of Astrolabe, the readers are recommended to read the original paper.

15.8 Summary

In this chapter, we studied the concepts of soft state replication and eventual consistency in depth. First, we learned about different consistency and data replication models, their costs and benefits. Then we looked into different techniques for software design of highly available applications where strict consistency models are relaxed to obtain inconsistent results of the computations. Next, we reviewed the concept of eventual consistency and studied how it is implemented in eBay's BASE using MQMIn a separate section, we reviewed salient features of the open-source AMQP for MQM. Of course, at some point inconsistent states must be resolved or converged towards consistent ones, and this is why we give an overview of some protocols for computing convergent states. Finally, we study how some of these protocols are used in Astrolabe, an influential experimental platform for hosting eventually consistent distributed applications and how hierarchical organization of the system improves it scalability.

Chapter 16

Cloud Computing at Facebook

Facebook is a multibillion dollar corporation that provides a social media platform for people to exchange information and it uses this information to make revenues by enabling other companies and organizations to advertise goods and services to people. The reason that we choose to study the Facebook's technical cloud platform is because it evolved into one of the most powerful and technically sophisticated environment whose inner-workings are published in peer-reviewed venues and it hosts its annual Facebook Developer Conference (F8) where engineers and managers presented the latest achievements in their implementations of cloud computing technologies[1].

Of course, Facebook's cloud is also an interesting example because of its scale: as of the end of 2018, Zephoria Digital marketing reports that there are over 2.27 billion active Facebook users monthly with approximately 1.15 billion mobile and 1.49 billion desktop daily active users, and it translates into 13,310 mobile and 17,245 desktop users per second[2]. Users post 510,000 comments, they update 293,000 statuses and upload 136,000 photos every minute. This scale intensity translates into the revenue of approximately $13.2 billion in the first quarter of 2018, which translates into $34 per Facebook user. Over 80 million small and medium businesses created web pages to represent themselves on Facebook. The world of the Facebook cloud organization is broken into regions with each region hosting several datacenters.

Interestingly, one key measure of information consumption is the *engagement rate*, which means users reactions, shares, comments, and some clicks on links, videos and images. Engagement rate is calculated as the number of engaged users divided by the total reach of that post. The highest average engagement rate is for Facebook video posts that is close to 6.01% and it is almost two percentage higher than the average for all posts, where the engagement rate for photo posts is at 4.81% and the link posts' is at 3.36% and status posts' is at 2.21%. These different engagement rates show that different items have different levels of accesses and concurrency levels, which is important for determining the level of eventual consistency and synchronization to achieve the convergent states.

[1] `https://www.f8.com`

[2] `https://zephoria.com/top-15-valuable-facebook-statistics`

In this chapter, we reconstruct the internal operations of the Facebook cloud using peer-reviewed publications by Facebook engineers and scientists at competitive scientific and engineering conferences. First, we will use a presentation of engineering managers Mssrs. O'Sullivan and Legnitto and Dr. Coons who is a software product stability engineer on the developer infrastructure at Facebook's scale[3]. Also, we include information from an academic paper on development and deployment of software at Facebook authored by Prof. Dror Feitelson at Hebrew University, Israel, Dr. Eitan Frachtenberg a research scientist at Facebook, and Kent Beck, an engineer at Facebook [48]. Next, we will study what consistency model is used at Facebook and how eventual consistency is measured at Facebook. After that, we will review configuration management at Facebook and learn about its tool called Kraken for improving performance of web services. Finally, we will study an in-house built VM called HipHop VM for executing the code in *Personal Home Pages or PHP Hypertext Preprocessor (PHP)* pages on the Facebook cloud.

16.1 Facebook Developer Infrastructure

It is not published what number of physical servers Facebook deploys and this number changes over time. However, a back of the envelope estimate can give us an idea. Each datacenter hosts 50,000 to 80,000 servers and there are up to 10 datacenters in each availability zone with up to five availability zones in each region. These numbers are approximate, but they allow us to make make an educated guess that with five regions there may be up to 20 million servers running various distributed objects at any given time. Over 1,000 software engineers write software for three distinct platforms (i.e., the PHP back end, iOS, and Android OS) and this software is deployed on these servers as well as users' smartphones. It is somewhat unprecedented in the history of software engineering that the software developers are also the users of their own software that gets deployed in the cloud infrastructure controlled by these developers and at this massive scale. In many ways, the developer infrastructure at Facebook is also a large distributed cloud-based applications where developers can be viewed as distributed objects [48].

In this abstraction, software developers are distributed objects that update other distributed objects that store the source code of some other distributed objects that provide services to Facebook users. Once updated, the source code is compiled and packaged into distributed objects and tests are run to determine if they behave as desired. If the tests fail, the notification messages are sent to the developer objects that update the source code with fixes. This process continues until the tests pass, and then these distributed objects are deployed on the servers in the Facebook cloud, the entire process is fully automated. The granularity of code updates is small with incremental functionality additions. This automated development process is often referred to as *continuous integration (CI) and continuous delivery (CD)* and it is widely used in many companies besides Facebook.

Source code objects are stored in Git repositories, a distributed source code control

[3] `https://www.youtube.com/watch?v=X0VH78ye4yY`

system. One Git repository is used per platform at Facebook, where its engineers run more than one million source control commands per day with over $100,000$ commits per week. Facebook cloud servers are used to store source code objects in a source control system, first SVN, then GIT, and later, Mercurial. This is important, since with approximate frequency of one commit per second, the development infrastructure must provide quick operations, so that developers do not waste their time on operations that result from concurrent changes and delays in merging them.

Suppose that ten developers modify the source code of the same application and commit and push their changes at the same time. Thus, these concurrent updates are likely to be in conflict, since these changes may be applied to the same statements and operations. Naturally, the operation push fails and the developers must merge their changes and repeat the commit/push. Some of them may do it faster than the other ones, leaving the latter to repeat the same situation to continue to pull the updates, merge, commit/push, By implementing server side rebases, pushes are simply added to a queue and processed in order, ensuring developers don't waste time getting stuck in a push/pull loop while waiting for an opportunity to push cleanly. The majority of rebases can be performed without conflict, but if there is then the developer gets notified to manually handle the merge then submit a new push. It also helps that they have a very intensive automated testing suite (also discussed in that video) so they can have greater confidence in the integrity of their automated merging.

Facebook web-based application is highly representative of the development and deployment process for cloud-based companies and organizations. To be competitive, it is important for businesses to try new features to see how customers like them and then to provide feedback quickly to developers and other stakeholders. The idea is to split the users of a cloud-based application into two groups, A and B where the users in the group A is the control group that keeps using the baseline application and the users in the group B is the treatment group that is exposed to new features of the same application. This process is called A/B testing [44]. Various measurements are collected from both groups and then their are compared statistically to determine if these measurements improved w.r.t. certain predefined criteria. If the improvement is deemed statistically significant, then the new features are rolled out to all users.

Even though we described the development process at Facebook at a high level, three key differences are important between cloud-based and individualized development/deployment. First, it is heavily automated at a large scale with monitoring and execution data collection and analysis that provides real-time feedback on various aspects of the (non)functional features. Second, users do not install or have access to the server-side code – they are limited to the functionality presented to them by the UI. Moreover, often they are not aware that the functionality changed or that there was a new release of the application, since it may improve the performance and availability without adding any new functional features. Finally, the time is significantly shortened between identifying new requirements and deploying their implementation to billions of users on millions of servers. Achieving that with monolithic individualized customer installations and deployments would be very difficult.

Here we are, at a point where the cloud infrastructure is not used just to deployment of the applications, but it is itself a service, a set of distributed object whose performances are monitored and controlled and whose exposed interfaces enable ser-

vice personnel to optimize their work to fine-tune application services to the customer base at a much higher response rates, where more jobs are created for fine-tuning the infrastructure to enable developers to write application code to provide better services.

16.2 Measuring Existential Consistency

Rene Descartes once stated: "if something exists, it exists in some amount. If it exists in some amount, then it is capable of being measured." Applying this statement to eventual consistency means counting stale data reads in the operations of a cloud-based application as a proportion of the total reads of all data objects. Ideally, measuring eventual consistency can improve health monitoring of a cloud-based application and determine how to improve its consistency. However, a fundamental problem in measuring eventual consistency is that there is no global clock in a true asynchronous distributed system. Attempting to assign clock processes to different distributed objects requires their synchronization that can be achieved with messages sent to these clock processes, which execute at different speeds and these messages have some unpredictable latencies. Making the application synchronous defeats the purpose of having eventual consistency in the first place. Hence, the problem of measuring eventual consistency is a difficult to accomplish in real-world distributed cloud applications.

In Facebook, an experimental approach was designed to obtain rough estimation of the eventual consistency based on sampling log data of operations performed on the objects described by the Facebook graph data model [92]. In this graph data model, the vertexes represent data elements (e.g., a user, a message post, a response to a message post, an uploaded image) and edges represent relations or associations between the data element. Some edges are bidirectional (e.g., two users who follow each other) and others are strictly directional (e.g., a response to a message post). These objects are stored in a relational database that is replicated with one master copy in some region and it is row-based sharded across multiple servers within the other regions. The replicated database is based on the master/slave architecture where each shard has a copy located in its master database and its replicas are in slave databases, which are updated asynchronously. Users do not access the databases directly, instead they interact with multiple web servers that are deployed in each datacenter.

Every datacenter in each region exposes web server endpoints to users who submit their requests using the Internet. The web servers interact with leaf caches, which are sets of high-speed access storages that store shards using some kind of a caching scheme. Leaf caches obtain their shards from the root cache, which in turn obtains shards from the database in each region. When read requests arrive, they are serviced from the leaf caches with a high hit ration, however, when a cache miss occurs, then the data is obtained either from the root cache or pulled from the database. Thus, with a high hit cache ratio, read requests are serviced with a low response time.

When a write request arrives, its data is written synchronously into a corresponding leaf cache and from the it is propagated to the root cache. If it is a slave region, then the data from the root cache is propagated to the root cache in the master region where it is written into the master database. After that, the master and slave databases are synchronized at a later time. Once a slave region database is updated, the newest data

is replicated in the caches along the cache hierarchy tree.

Clearly, this architecture does not provide a strict consistency, which may not be required at all if we think about the business model of Facebook and social media applications in general. When users access objects, e.g., a post, they want to see a logical ordering of reads and writes on this object. Simply put, suppose users from different regions reply to the same post, i.e., the read the original post and write reply messages to it. Thus, for each object, it is important that updates are given in the sequential order in which they occur. It is also desirable that within each region the replies to the posts are seen in the sequential order, i.e., the reads see the latest writes on the specific object, however, the overall consistency of all objects in all regions and their caches, is eventual. To summarize, within Facebook per-object sequential consistency is used where a total order over all clients' requests to each object, so that clients always see the newer version of this object, even though different clients obtain different states of the object. Leaf caches and regions provide read-after-write consistency where subsequent reads to some write in the same cache retrieve the value updated by that write command.

In order to measure eventual consistency, log messages with timestamps are analyzed to determine if stale values are returned in response to read requests. Consider two situations that are shown in Figure 16.1. The nodes designate **r**ead or **w**rite operations on some object where the subscript index shows the order of the operation and the superscript shows the value written into or read from the object. The solid directed edges designate the sequential order of the operations whereas the dashed edges show the actual order as it is established by the values returned by read operations matching the previous write operations. For example, the order established by timestamps on the operations is that the first write operation put value 5 into the object followed by the next write operation that replaced this value with 7 as shown in Figure 16.1a. The next operation, read occurred after the last operation write, however, it retrieved value 5 from the object, which is likely to have been written by the penultimate operation write. This causal value dependency is shown with the dashed arrow from the node labeled w_1^5 to the node labeled r^5. Together these nodes form a loop that is shown as a dashed circle in the enclosed space. The presence of a loop is a strong indication that the stale data is read as a result of delayed updates in the eventual consistency model.

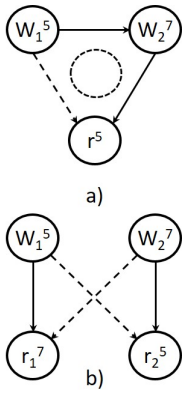

Figure 16.1: Loop formation.

Similar but more complicated situation is shown in Figure 16.1b where the loop is formed the following way: $w_1^5 \rightarrow r_2^5 \rightarrow w_2^7 \rightarrow r_1^7 \rightarrow w_1^5$. It happens as a result of two concurrent writes that happen at the same time and the following concurrent reads. As a result, there is no causal dependencies between the write nodes and there are no dependencies between the read nodes. However, there are causal dependencies between write and read nodes that form the loop that indicates the presence of the stale data reads. Detecting these loops enables pinpointing propagation of the stale data and determining the degree of eventual consistency in the system.

Of course, measuring the level of consistency by relying on a subset of log data introduces a certain level of imprecision. Relying on timestamps in log messages means

that the clocks must be well-synchronized, and in Facebook the clock drift was estimated approximately 35ms and it is accounted for in the consistency calculating algorithm. The other problem is when newer values are equal to the previous values – it is not possible to determine the dashed dependency edges. Regardless, the experiment within Facebook showed that out of hundreds of millions of captured read/write operations, only a few thousand are registered as loop anomalies, or between 0.001% and 0.0001% of the total requests. Interestingly, only approximately 10% of the objects experience both reads and writes and they are the sources of inconsistencies. With a very small percentage of the total number of objects leading to inconsistency, it is unclear if serious remediation measures are needed.

Based on this experiment, Facebook introduced the concept of ϕ(P)-consistency for a set of replicas, P that measures the percentage of these replicas that return the same most recently written values for the same read request. Moreover, ϕ(P)-consistency is divided into ϕ(G)-consistency and ϕ(R)-consistency. The former measures the consistency of all cache replicas on the global scale and the latter does so only for some **R**egion. Using these measures allows engineers at Facebook to find undesired latencies and problems with network configurations more effectively. For more information, readers should be directed to the original paper [92].

16.3 Configuration Management

Large cloud management applications consist of millions of *lines of code (LOC)* and tens of thousands of LOC for configuration artifacts in a variety of formats, most popular of which are XML, JSON, and key/value pairs that are expressed as assignments, e.g., `CONF_VAR_NAME = someValue`. Unlike input parameters of applications where values are passed to function calls and then they are used in the implemented algorithms in these functions to compute some results, configuration parameters specify how components interact at the architecture level, e.g., the configuration parameter `innodb_checksum_algorithm` for the configuration file `backup-my.cnf` for popular RDBMS MySQL specifies what algorithm to use for computing the checksum value. Of course, there is a certain overlap between the input parameter values and configuration file definitions; however, changing the values in configuration files does not require recompilation of a portion or the entire codebase, which often requires significant resources and time in addition to redeploying newly built software artifacts.

Automating configuration management in the cloud infrastructure is important because configuration parameters are specified in tens of thousand of configuration files. One configuration parameter can control the execution of multiple independent code fragments in different components and combinations of the values of different configuration parameters may lead to states in different applications that trigger crashes and severe worsening of the system's performance. One example in Facebook describes how changing a single configuration parameter enable a code fragment whose concurrent execution lead to a subtle race condition that resulted in applications' crashes [147].

The simplicity of defining and modifying configuration parameter values is entangled with the devilish complexity of determining how these changes affect the cloud infrastructure and millions of applications that run on it. A programmer can simply

open a text file with key/value pair assignments and change a value of some configuration parameter – the repercussions of this change may manifest itself in component failures, which are separated from the change spatially (e.g., in an unrelated component running on a remote server) or temporally (e.g., after a long period of system execution time). Configuration files are pervasive in the cloud datacenters and they are used to manage the following aspects, among many others.

Product features are released to groups of customers frequently their release is controlled by switching values of the appropriate configuration parameters.

A/B testing is similar to releasing new product features where live experiments are performed with selected groups of users, as it is controlled by selected configuration parameters.

Network traffic is controlled by configuration parameters that switch on/off routing of messages or inclusion of features in the product that demand additional message delivery and processing.

Load balancing config parameters switch on/off virtual routers and change the configuration of the network topology to improve the overall performance of the cloud infrastructure.

Smart monitoring is adjusted by a multitude of configuration parameters that determine the monitoring granularity and remedial actions depending on a certain pattern of data.

A key idea of the automated cloud-based configuration management is that while it is easy for an engineer to change a value of some configuration parameter, it is often equally easy to make a mistake when evaluating the impact of this change on the entire cloud infrastructure. For example, it is documented that an incorrectly set configuration parameter value that determines which cache machine to send a request to led to cache invalidations that potentially result in a higher number of cache misses and the increased performance penalties. To prevent these errors, the job of changing configuration parameters is automated with a configuration compiler that can be viewed as an extension of an IDL compiler for RPC.

Thus, a request to add/remove/modify a configuration parameter is implemented as a program with an RPC to a distributed object, which encodes relationships between the target configuration parameter and other configuration parameters. For example, consider a situation where a configuration parameter named `SPOTMARKET` defines the minimum price per hour for running a VM, so that those VMs for which owners pay less than this minimum price will be terminated after some preset timeout. However, it may automatically reset the parameter `SAVEVMSTATE` to ensure that when the price of running a VM drops below the threshold specified in the parameter `SPOTMARKET`, the terminated VMs will restart again from the instructions at which they were stopped. To account for dependencies among different configuration parameters, a special configuration compiler analyzes dependencies among configurations for a given parameter and modifies the values of the dependent configuration parameters accordingly. Doing so automates the manual process of changing configuration parameters with additional

validations that enable engineers to avoid many mistakes in a complex cloud environment with many interdependent processes.

Apache Thrift is used to specify configuration data types as an interface of the remote configuration object. For example, one configuration data element could be a financial instrument for trading, e.g., bonds, and some other configuration data element is a protocol used to send trading messages for a given financial instrument, e.g., `FIX`. In some cases, certain financial instruments can be traded using only specific protocols whereas in some other cases some protocols are disallowed for being used with specific financial instruments. These constraints are encoded in a special program maintained by cloud service personnel. This program contains reusable routines for validating combinations of the configuration option values. Reusable configurations are written in `cinc` files, e.g., if the chosen financial instrument is bond, then the configuration variable for protocol is automatically assigned the value `FIX`. Programmers specify their changes to configuration options in separate files in a given language, i.e., Python, and a different program called Configerator Compiler verifies the submitted configuration changes against the validation program and `cinc` files to determine that these configuration changes do not violate any constrains. Once verified, the configuration changes are applied to the system.

Configuration management is a big part of services that cloud organizations provide to their customers. Not only do cloud providers configure sophisticated software packages for hundreds of thousands of their customers, but also they configure various software modules internally that are used by these customers. Using the RPC approach to perform automated centralized configuration management enables cloud providers to reduce the downtime and improve the availability of the cloud services.

16.4 Managing Performance of Web Services

A rather informal and much debated 15-second rule states that close to two third of visitors spend less than 15 seconds waiting for a website to appear in their browsers before moving on. Various performance metric advisories state that the website loading time is the most important metric that determines the probability that a customer leaves a website. Assuming that most customers have reasonably fast network connections, it is therefore very important that the latency of the cloud-based web services is reduced to a very small value. Often, applications are performance tested before being released into production and doing so is expensive without any guarantees of the success.

A goal of performance testing is to find performance problems, when an *application under test (AUT)* unexpectedly exhibits worsened characteristics for a specific workload [100, 157]. For example, effective test cases for *load testing*, which is a variant of performance testing, find situations where an AUT suffers from unexpectedly high response time or low throughput [17, 78]. Test engineers construct performance test cases, and these cases include actions (e.g., interacting with GUI objects or invoking methods of exposed interfaces) as well as input test data for the parameters of these methods or GUI objects [76]. It is difficult to construct effective performance test cases that can find performance problems in a short period of time, since it requires test engineers to test many combinations of actions and data for nontrivial applications [65].

Developers and testers need performance management tools for identifying performance problems automatically in order to achieve better performance of software while keeping the cost of software maintenance low. Performance *bottlenecks* (or *hot spots*) are phenomena where the performance of the entire system is limited by one or few components [4, 11]. In a survey of 148 enterprises, 92% said that improving application performance was a top priority [134, 158]. The application performance management market is over USD 2.3Bil and growing at 12% annually, making it one of the fastest growing segment of the application services market [16, 57]. Existing performance management tools collect and structure information about executions of applications, so that stakeholders can analyze this information to obtain insight into performance; unfortunately, identifying performance problems automatically let alone correcting them is the holy grail of performance management. Performance problems that result in productivity loss approaching 20% for different domains due to application downtime [67].

When a user enters the address "`www.facebook.com`" in her browser, it is resolved to an IP address of some *Point of Presence (POP)* LB server in the geographic proximity to the user. The LB server multiplexes the request from the user to a regional datacenter whose LB entry servers multiplex this request further to a cluster within the datacenter, which in turn uses its LB to direct this request to a server within the cluster. When making multiplexing choices, LB servers may use sophisticated algorithms, however, in reality situations occur when multiple datacenters can serve a specific request from a given user and the difference is that within a datacenter with a slightly higher network latency a server may be available whose hardware configuration will result in much smaller computation latency. That is, it may take longer to submit a request from a user to a specific server, however, this server will perform a computation for this request much faster compared to a server in the datacenter that can be reached faster because of its slightly better geographic proximity.

Kraken is a automatic performance tool created and used at Facebook that uses feedback-directed loop to change the traffic between regions to balance the load [153]. We encounter feedback-driven loops in our devices everywhere - automatic systems in our houses uses sensors to measure the temperature to determine when to turn on and off the air conditioner or to measure the level of the visibility to change the intensity of the street lights. In Kraken, this idea is applied to collect various performance measurements such as the loads of the servers and various latencies among many others to change the distribution of the customer request messages among the regions and datacenters and their clusters. The latter is achieved by assigning weights to channels between POPs, regions, datacenters, and clusters and using LBs to change these weights in response to collected measurements, thus creating the feedback-directed loop between performance indicators of the cloud and the distribution of the load.

In the nutshell, Facebook's Kraken works like the following way. The cloud monitoring subsystem collects diverse metrics on CPU and memory utilization, error rates, latency, network utilization, write success rate, exception rate, lengths of various queues, retransmit rates, and object lease count for in-memory data processing software. Of course, various metrics are added over time. These metrics and their timestamps are collected at predefined time intervals and they are stored in in-memory high-performance database called Gorilla [114] and its open-source implementation

called Beringei is publically available[4]. Once Kraken determines that an imbalance exist in the cloud where some datacenters or their clusters are getting overloaded whereas other subsystems are underutilized, the weights are modified to change the distribution of the traffic. As a result, new data metrics is collected and the feedback loop continues.

At a lower level of the implementation, Facebook uses Proxygen[5], a library for creating LB-based web servers. A Proxygen-based web server ingests a cloud network topology description from an external configuration file that contains weights for channels that connect POP, datacenters, clusters and web servers. Every minute Kraken collects the metrics, aggregates the values, and then in the following minute issues corrective actions for the weights based on the aggregated metric values. Once the weights are changed, the traffic will shift.

It is noteworthy that a core assumption behind the use of Kraken is that the servers in the cloud datacenter must be stateless. Consider a web server that must record the state of the previous computations and use it in the subsequent computations. To use a proper portion of the recorded state for a specific computation, stateful servers establish *sessions* that define a logically and physically bounded exchange of messages. As a result, requests for stateful computations must be routed to particular servers in datacenters because they have specific states. Session can be long and they can encapsulate many RPC computations. To ensure that all session requests are forwarded to a specific server, the notion of the *sticky session* is used where RPCs are marked as part of a particular session. As a result, load balancers will have to direct sticky session requests to designated servers and it will interfere with Kraken's traffic modifications by directing RPCs to arbitrary servers in different datacenters based on the metrics.

16.5 The HipHop VM

Facebook content production and delivery programs are written using PHP – it stands for Personal Home Pages or it is a recursive acronym for PHP: Hypertext Preprocessor. PHP programs contain executable code in a Java-like imperative object-oriented language embedded into HTML. This language has been extended over years with many sophisticated features, e.g., traits, exceptions, generators. This code is evaluated on the server side in response to a client's request and the resulting HTML page is sent to the client. Using server-side scripting is a common mechanism for deploying and executing PHP programs where a web server has a PHP extension to parse PHP programs in response to request from clients, execute code embedded in these programs, and serve the results of these executions to clients.

Unlike statically typed compiled languages like Java or Scala, PHP is very easy to learn, because programmers do not have to understand the constraints imposed by a type system. It is also expressive and there are reports that ten-year old children can learn to write basic PHP code. Unfortunately, these attractive features of PHP are balanced with its downsides: it is a scripting interpreted language, its programs often fail at runtime for some basic type mismatch errors, and it is not well-suitable for large-scale development, since its modularization facilities are somewhat lacking

[4] `https://github.com/facebookarchive/beringei`
[5] `https://github.com/facebook/proxygen`

```
function compare($v1, $v2) { return $v1 == $v2 ? 0 : 1; }
```

Figure 16.2: HHVM tracelet that compares two input values.

robust enforcement of data encapsulation. All in all, PHP programs are difficult to optimize and they are much slower in execution and error prone. However, given a large PHP codebase at Facebook, it is not a sensible business solution to rewrite it in some more efficient language, especially given that maintenance and new development is required to add new features and fix bugs in the existing deployment of Facebook. If the codebase rewrite is not possible, then a solution is to create a new execution platform for PHP programs to address their performance drawbacks.

The new execution platform is a process-level VM called *HipHop VM (HHVM)* that abstracts away the runtime interpretation of the PHP code by introducing a new set of instructions called *HipHop Byte Code (HHBC)* [2]. A set of tools come with the HHVM to compile PHP programs to HHBC and optimize the resulting bytecode programs using *Just in Time (JIT)* compiler. Recall from Section 7 that a process VM does not require a hypervisor, it can be a library loaded in the memory. The untyped, dynamic nature of PHP programs makes it difficult to determine exact types of data and their representations in PHP programs at compile time, hence the programs are optimized by the HHVM during runtime. As the HHBC program keeps executing with different input data, the HHBC is optimized based on the types of data detected during the execution, thus making programs run faster the more times they are executed.

Financially, it is likely a sensible solution. Suppose that a new execution platform can increase the performance of the cloud by 20%. Since Facebook uses approximately 20Mil servers, the improvement in the performance and efficiency of the executing PHP programs may result in freeing an equivalent of a few millions of servers resulting in a saving of tens of millions of USD per year not including the service and the maintenance costs. Hence, the development of a new VM is not driven by pure curiosity or simply the desire of software engineers to create some other platform, but a business need to optimize the resources and the economics of the investment.

HHVM optimizes PHP programs' bytecode using the abstraction of a bytecode unit called a *tracelet* that can be viewed as a box with a single input multiple outputs. As the data flows into the input entry, it is processed by the HHBC and the output data flows out of the outputs. If the types of the input values were known at compile time, the HHVM compiler could determine their runtime representations and optimize the operations. Instead, the JIT compiler discovers the types of the data in a tracelet at runtime and it generates multiple instances of the tracelet with guards that describe the types of the input and output data. When a tracelet is invoked later, the JIT compiler determines the types of the input data and it will find the appropriate version of the tracelet for these types using the guards.

Consider an example of a tracelet in Figure 16.2 that compares the values of two input variables and returns zero if they are equal and one otherwise. The types of the input variables, $v1 and $v2 are not known. When the function `compare(1,2)` is invoked, the JIT compiler determines that the type of the input data is `integer` and it uses HHBC instructions that deal with this type in the generated tracelet and correspondingly it creates the guards that designate the generated tracelet for the integer

inputs. Likewise, when the function `compare(''howdy'',''comrade'')` is invoked, the JIT compiler determines that the input data type is `string` and it will generate a new version of the tracelet with the guards for the input data type `string`. Continuing with the chain of invocations among tracelets, the JIT compiler updates the flow of invocations of tracelets with the corresponding guards. When the invocation occurs with new data types or the flow changes based on the newly activated conditions, the bindings between tracelets are updated at runtime, making the tracelet model fluid and re-adjusted to the use of the programs. HHVM is one of the successful examples of using the abstractions from dynamic binary translation methods.

HHVM is publicly available[6] and at the time of writing this edition of the book, HHVM phases out its support for PHP and it is focused on Hack[7], a hybrid language that combines dynamic and static typing and it is built specifically for HHVM. Discussing Hack is beyond the scope of this book and interested readers can obtain all needed information directly from Facebook Open Source.

16.6 Summary

This chapter was dedicated to the organization and operation of the Facebook cloud. We live in an amazing time when we can determine how technical solutions are engineered at large organizations simply by reading technical papers that they published. We reviewed the development infrastructure at Facebook using which the code is created and tested before deployed for use by billions of users. Setting aside the business goal of Facebook, the scale of cloud deployment is truly breathtaking: billions and billions of users are served every day to enable a continuous and non-interrupted flow of information. Interestingly, the software development lifecycle for creating and deploying software in the cloud differs from the classic waterfall/spiral models, since the A/B testing process is embedded into the development mode. Releasing software to its users who test the software by using it may be viewed as the alpha- or the beta-testing preproduction phase except from the users' point of view the software is in production. In a way, this way of rapid deployment enables software engineers to add new features to large software applications at a fast pace.

Next, we reviewed the Facebook algorithm of measuring the eventual consistency and we learned how different levels of eventual consistency are used to adjust caching and the data flow to make data more consistent while preserving the availability of the cloud services. We analyzed the configuration management in the cloud as an important mechanism to synchronize different components into a correctly assembled software system. The introduction of the configuration compiler called Configerator treats configurations as distributed objects with the RPC-based constraint enforcement mechanism. Finally, we studied Kraken and HPVM, feedback-directed performance management system with an intermediate bytecode translator and optimizer.

[6] `https://hhvm.com`
[7] `https://hacklang.org/`

Chapter 17

Conclusions

All good things come to an end and this book is not an exception. We have covered a lot of ground components of the new engineering technology called cloud computing. Its fundamental difference from other related technologies lies in managing computing resources, dynamically at a wide scale and putting the cost on these resources based on demand for them. As such, cloud datacenters go well beyond providing distributed RPC-based services – they are also revenue management systems akin to the airline industry where resources represent seats on an airplane. Of course, passengers are not applications in the sense that they are not forcibly ejected from their seats when the demand changes, well, not by all airlines. But the idea of balancing the load among resources based on the elastic demand of applications and the willingness of the applications' owners to pay for these resources is unique in the short history of software and systems engineering at a wide scale.

Predicting how technologies will evolve is an impossible task. It has been proven repeatedly that making these predictions is a fool's errand. Yet it is likely that the technology will be improved in the near future incrementally, not by changing everything overnight. Major trends include computing consistent results in highly available systems in the presence of partitioning, highly efficient utilization of cloud resources, and revenue management to determine dynamic price ranges for resources based on their utilization and the customer demand. We will discuss these trends below.

There is a decade-old debate about the CAP theorem and its influence on the development and deployment of cloud-based applications. Whereas the CAP theorem establishes theoretical limits on selecting two out of three properties, i.e., consistency, availability or partitioning, modern cloud organizations become increasingly reliable w.r.t. hardware and network faults. It is expected that with multiple redundant high-speed network connections and physical servers the replicas will be synchronized very fast and the eventual consistency will be minimal thus leading to highly consistent and available cloud computing.

Intelligent load balancers will lead to significant improvements of utilization of cloud services and to the decreased latency of computations. LBs will use machine learning algorithms and techniques to route requests from users to regions, datacenters, clusters and specific servers based on the collected runtime information like in

Facebook's Kraken. Of course, these algorithms take time and resources, however, as they will become more efficient and more precise, their benefits will outweigh their costs resulting in learning LBs that will improve the performance and efficiency of cloud datacenters.

Cloud computing requires users to pay for used resources, so this cost dimension adds emphasis to better elasticity and smart revenue management. As we learned in this book, perfectly elastic services are impossible to create because it is impossible to determine the exact performance of nontrivial software applications. However, improving elasticity will result in smaller amounts of under- and over-provisioning of resources thereby increasing the value of cloud computing services to customers. Equally important it will be to determine prices of resources dynamically and automatically based on the changing users' demand. Revenue management is a complex science that is rooted in operation research, and the optimization of cloud resource pricing reflects its use and its value thus adding to making cloud computing available, efficient, and a default choice of deploying software applications for individual customers and large companies and organizations for many decades to come.

Bibliography

[1] *Oxford Living Dictionary*. Oxford University Press, 2017. Latency.

[2] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014. ACM.

[3] C. C. Aggarwal and P. S. Yu. *Privacy-Preserving Data Mining: Models and Algorithms*. Springer, 2008.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, 2003.

[5] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 251–260, New York, NY, USA, 1993. ACM.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[7] M. Albonico, J.-M. Mottu, and G. Sunyé. Controlling the elasticity of web applications on cloud computing. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 816–819, New York, NY, USA, 2016. ACM.

[8] M. Allman. An evaluation of xml-rpc. *SIGMETRICS Perform. Eval. Rev.*, 30(4):2–11, Mar. 2003.

[9] Amazon. Amazon web services. `http://aws.amazon.com`, May 2017.

[10] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo. Virtual cpu validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 311–327, New York, NY, USA, 2015. ACM.

[11] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, pages 170–194, 2004.

[12] E. Anderson, L. ling Lam, C. Eschinger, S. Cournoyer, J. M. Correia, L. F. Wurster, R. Contu, F. Biscotti, V. K. Liu, T. Eid, C. Pang, H. H. Swinehart, M. Yeates, G. Petri, and W. Bell. Forecast overview: Public cloud services, worldwide, 2011-2016, 4q12 update. Technical report, Gartner, Stamford, CT, USA, 2013.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[14] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[16] C. Ashley. Application performance management market offers attractive benefits to european service providers. *The Yankee Group*, Aug. 2006.

[17] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *ISSTA*, pages 44–57, New York, NY, USA, 1994. ACM.

[18] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[19] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, May 2004.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[21] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[22] M. A. N. Bikas, A. Alourani, and M. Grechanik. How elasticity property plays an important role in the cloud: A survey. *Advances in Computers*, 103:1–30, 2016.

[23] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.

[24] K. P. Birman. *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*. Texts in computer science. Springer, 2012.

[25] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984.

[26] P. Blobaum. *Parents of Invention: The Development of Library Automation Systems in the Late 20th Century*. Christopher Brown-Syed Santa Barbara, CA: Libraries Unlimited., Santa Barbara, CA, USA, 2014.

[27] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, OOPSLA '94, pages 341–354, New York, NY, USA, 1994. ACM.

[28] P. Bollen. Bpmn: A meta model for the happy path. 2010.

[29] P. C. Brebner. Is your cloud elastic enough?: Performance modelling the elasticity of infrastructure as a service (iaas) cloud applications. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 263–266, New York, NY, USA, 2012. ACM.

[30] E. Brewer. A certain freedom: Thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 335–335, New York, NY, USA, 2010. ACM.

[31] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[32] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10:70–10:93, Jan. 2016.

[33] R. N. Calheiros, A. N. Toosi, C. Vecchiola, and R. Buyya. A coordinator for scaling elastic applications across multiple clouds. *Future Gener. Comput. Syst.*, 28(8):1350–1362, Oct. 2012.

[34] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[35] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[36] CISCO. Cloud computing adoption is slow but steady. *ITBusinessEdge*, 2012.

[37] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[38] U. F. E. R. Commission. Assessment of demand and response advanced metering.

[39] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, Dec. 2002.

[40] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, 1989.

[41] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, Sept. 1985.

[42] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[43] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[44] A. Deng, J. Lu, and J. Litz. Trustworthy analysis of online a/b tests: Pitfalls, challenges and solutions. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 641–649, New York, NY, USA, 2017. ACM.

[45] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Inf. Process. Lett.*, 11(1):1–4, 1980.

[46] U. Drepper. What every programmer should know about memory, 2007.

[47] M. Eriksen. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 5:1–5:7, New York, NY, USA, 2013. ACM.

[48] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[49] C. Ferris and J. Farrell. What are web services? *Commun. ACM*, 46(6):31–, June 2003.

[50] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[51] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948–960, 1972.

[52] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[53] I. T. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. *CoRR*, abs/0901.0131, 2009.

[54] T. Friebel. How to deal with Lock-Holder preemption. Technical report.

[55] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data transfer matters for gpu computing. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ICPADS '13, pages 275–282, Washington, DC, USA, 2013. IEEE Computer Society.

[56] J. Gable. Enterprise applications: Adoption of e-business and document technologies, 2000-2001: Worldwide industry study. Research note, Gartner Inc., 2001.

[57] J.-P. Garbani. Market overview: The application performance management market. *Forrester Research*, Oct. 2008.

[58] F. Gens, M. Adam, M. Ahorlu, D. Bradshaw, L. DuBois, M. Eastwood, T. Grieser, S. D. Hendrick, V. Kroa, R. P. Mahowald, S. Matsumoto, C. Morris, R. L. Villars, R. Villate, N. Wallis, and A. Florean. Worldwide and regional public it cloud services 20122016 forecast. Technical report, IDC, Framingham, MA 01701 USA, 2012.

[59] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 316–326, New York, NY, USA, 1991. ACM.

[60] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[61] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[62] Google. Auto scaling on the google cloud platform. *Google Cloud Platform*, Oct. 2013.

[63] Google. Google cloud platform. `https://cloud.google.com`, May 2017.

[64] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[65] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 156–166, 2012.

[66] N. Griffeth, N. Lynch, and M. Fischer. Global states of a distributed system. *IEEE Transactions on Software Engineering*, 8:198–202, 1982.

[67] T. Y. Group. Enterprise application management survey. *The Yankee Group*, 2005.

[68] J. O. Gutierrez-Garcia and K. M. Sim. A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. *Future Gener. Comput. Syst.*, 29(7):1682–1699, Sept. 2013.

[69] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Commun. ACM*, 19(5):266–272, May 1976.

[70] J. Handy. *The Cache Memory Book*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[71] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX.

[72] J. Hill and H. Owens. Towards using abstract behavior models to evaluate software system performance properties. In *Proceedings of the 5th International Workshop on Advances in Quality of Service Management*, (AQuSerM 2011, New York, NY, USA, 2011. ACM.

[73] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[74] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[75] C. Huffman and V. Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2010.

[76] IEEE. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers, January 1991.

[77] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 85–96, New York, NY, USA, 2012. ACM.

[78] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *ICSM*, pages 125–134, 2009.

[79] P. R. Johnson and R. Thomas. Maintenance of duplicate databases, 1975.

[80] L. Kawell, Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work*, CSCW '88, pages 395–, New York, NY, USA, 1988. ACM.

[81] R. Kay. Pragmatic network latency engineering fundamental facts and analysis.

[82] J. Kim, D. Chae, J. Kim, and J. Kim. Guide-copy: Fast and silent migration of virtual machine for datacenters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 66:1–66:12, New York, NY, USA, 2013. ACM.

[83] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, 2014. USENIX Association.

[84] V. Kundra. Federal cloud computing strategy. *Office of the CIO, Whitehouse*.

[85] T. S. U. G. Lab. The stanford university graphics lab. *http://graphics.stanford.edu/projects/gpubench/*, 2011.

[86] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[87] B. W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating System Principles, SOSP 1983, Bretton Woods, New Hampshire, USA, October 10-13, 1983*, pages 33–48, 1983.

[88] A. N. Langville and C. D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.

[89] G. Lee. *Cloud Networking: Understanding Cloud-based Data Center Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[90] R. Lipton. PRAM: A scalable shared memory. Technical report, Princeton University, 1988.

[91] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[92] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[93] B. D. Lynch, H. R. Rao, and W. T. Lin. Economic analysis of microcomputer hardware. *Commun. ACM*, 33(10):119–129, Oct. 1990.

[94] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, Dec. 2013.

[95] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society.

[96] M. K. McKusick, G. Neville-Neil, and R. N. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[97] P. M. Mell and T. Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[98] H. Mendelson. Economies of scale in computing: Grosch's law revisited. *Commun. ACM*, 30(12):1066–1072, Dec. 1987.

[99] R. Mian, P. Martin, F. Zulkernine, and J. L. Vazquez-Poletti. Towards building performance models for data-intensive workloads in public clouds. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 259–270, New York, NY, USA, 2013. ACM.

[100] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2009.

[101] T. K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, 2005.

[102] D. Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, Jan. 1993.

[103] MSAzure. Microsoft azure. `https://azure.microsoft.com`, May 2017.

[104] MSAzure. Sizes for cloud services. `https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-sizes-specs`, May 2017.

[105] MSAzure. Sizes for windows virtual machines in azure. `https://docs.microsoft.com/en-us/azure/virtual-machines/virtual-machines-windows-sizes?toc=\%2fazure\%2fvirtual-machines\%2fwindows\%2ftoc.json`, May 2017.

[106] S. Mullender, editor. *Distributed Systems (2Nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[107] Y. V. Natis. Service-oriented architecture scenario. Research Note AV-19-6751, Gartner Inc., Apr. 2003.

[108] Y. V. Natis and R. V. Schulte. "service oriented" architectures, part 1. Research Note AV-19-6751, Gartner Inc., Apr. 1996.

[109] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, Aug. 1991.

[110] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen. Cost-effective cloud hpc resource provisioning by building semi-elastic virtual clusters. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 56:1–56:12, New York, NY, USA, 2013. ACM.

[111] J. O'Hara. Toward a commodity enterprise middleware. *Queue*, 5(4):48–55, May 2007.

[112] D. C. Oppen and Y. K. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Trans. Inf. Syst.*, 1(3):230–253, July 1983.

[113] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[114] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, Aug. 2015.

[115] D. Perez-Palacin, R. Mirandola, and M. Scoppetta. Simulation of techniques to improve the utilization of cloud elasticity in workload-aware adaptive software. In *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*, ICPE '16 Companion, pages 51–56, New York, NY, USA, 2016. ACM.

[116] M. Perrin, A. Mostefaoui, and C. Jard. Causal consistency: Beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 26:1–26:12, New York, NY, USA, 2016. ACM.

[117] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. `http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc`, 2009.

[118] D. B. Petriu and M. Woodside. Software performance models from system scenarios. *Perform. Eval.*, 61(1):65–89, June 2005.

[119] M. Pharr. *Network Analysis and Troubleshooting.* Addison-Wesley, Dec 1999.

[120] M. Pharr. GPU *Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley, March 2005.

[121] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[122] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[123] G. V. Post. How often should a firm buy new pcs? *Commun. ACM*, 42(5):17–21, May 1999.

[124] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 9–22, New York, NY, USA, 2013. ACM.

[125] D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[126] J. Protic, M. Tomasevic, and V. Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[127] W. E. Riddle, J. C. Wileden, J. H. Sayler, A. R. Segal, and A. M. Stavely. Behavior modelling during software design. In *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, pages 13–22, Piscataway, NJ, USA, 1978. IEEE Press.

[128] M. Rost, C. Fuerst, and S. Schmid. Beyond the stars: Revisiting virtual cluster embeddings. *SIGCOMM Comput. Commun. Rev.*, 45(3):12–18, July 2015.

[129] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[130] C. Sapuntzakis and M. S. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 10–10, Berkeley, CA, USA, 2003. USENIX Association.

[131] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 377–390, Berkeley, CA, USA, 2002. USENIX Association.

[132] F. B. Schneider. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.*, 4(2):125–148, Apr. 1982.

[133] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[134] C. Schwaber, C. Mines, and L. Hogan. Performance-driven software development: How it shops can more efficiently meet performance requirements. *Forrester Research*, Feb. 2006.

[135] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[136] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using functional programming within an industrial product group: Perspectives and perceptions. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 87–92, New York, NY, USA, 2010. ACM.

[137] P. Scott. *Shader Model 3.0, Best Practices*. nVidia, 2007.

[138] M. Seshadrinathan and K. L. Dempski. High speed skin color detection and localization on a GPU. In *Proceedings of Eurographics 2007*, pages 334–350, Prague, Czech Republic, 2007. Eurographics Association.

[139] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala. page 15, 2013.

[140] M. Shiraz, E. Ahmed, A. Gani, and Q. Han. Investigation on runtime partitioning of elastic mobile applications for mobile cloud computing. *J. Supercomput.*, 67(1):84–103, Jan. 2014.

[141] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5, 2007.

[142] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[143] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[144] S. Soltesz, M. E. Fiuczynski, L. Peterson, M. McCabe, and J. Matthews. Virtual doppelganger: On the performance, isolation, and scalability of para- and paene-virtualized systems.

[145] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi. Rpc chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 277–290, Berkeley, CA, USA, 2009. USENIX Association.

[146] S. Suneja, C. Isci, E. de Lara, and V. Bala. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 133–146, New York, NY, USA, 2015. ACM.

[147] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 328–343, New York, NY, USA, 2015. ACM.

[148] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 286–297, New York, NY, USA, 2017. ACM.

[149] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[150] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[151] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.

[152] R. vanderMeulen. Gartner says by 2020 "cloud shift" will affect more than $1 trillion in it spending. `http://www.gartner.com/newsroom/id/3384720`, May 2017.

[153] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, Savannah, GA, 2016. USENIX Association.

[154] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[155] V. S. Verykios, E. Bertino, I. N. Fovino, L. P. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in privacy preserving data mining. *SIGMOD Rec.*, 33(1):50–57, Mar. 2004.

[156] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, Nov. 2006.

[157] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, 2000.

[158] N. Yuhanna. Dbms selection: Look beyond basic functions. *Forrester Research*, June 2009.

[159] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[160] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[161] Z. Zhang, C. Wu, and D. W. Cheung. A survey on cloud interoperability: Taxonomies, standards, and practice. *SIGMETRICS Perform. Eval. Rev.*, 40(4):13–22, Apr. 2013.