DISTRIBUTED SYSTEM

Dr. A. Jayachandran
Ramesh Chandra Tripathi

ALEXIS PRESS
JERSEY CITY, USA

DISTRIBUTED SYSTEM

DISTRIBUTED SYSTEM

Dr. A. Jayachandran
Ramesh Chandra Tripathi

ALEXIS PRESS

Published by: Alexis Press, LLC, Jersey City, USA
www.alexispress.us

© RESERVED

This book contains information obtained from highly regarded resources.
Copyright for individual contents remains with the authors.
A wide variety of references are listed. Reasonable efforts have been made
to publish reliable data and information, but the author and the publisher
cannot assume responsibility for the validity of
all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted,
or utilized in any form by any electronic, mechanical, or other means,
now known or hereinafter invented, including photocopying,
microfilming and recording, or any information storage or retrieval system,
without permission from the publishers.

For permission to photocopy or use material electronically
from this work please access alexispress.us

First Published 2022

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication Data

Includes bibliographical references and index.

Distributed System by *Dr. A. Jayachandran, Ramesh Chandra Tripathi*

ISBN 978-1-64532-891-9

CONTENTS

Chapter 1. Introduction of the Distributed System 1
— Dr. A. Jayachandran

Chapter 2. Supporting Resource Sharing 6
— Dr.S.P.Anandaraj

Chapter 3. Degree of Distribution Transparency 9
— Ms.Bhavya

Chapter 4. Scaling Techniques 15
— Dr. Pallavi R

Chapter 5. An Elaboration of the Cloud Computing 22
— Dr. Blessed Prince

Chapter 6. Distributed Transaction Processing 25
— Dr. Jayavadivel

Chapter 7. Context Awareness 29
— Dr. Gopal K Shyam

Chapter 8. Architecture of Distributed System 35
— S.poornima

Chapter 9. Resource Based Architectures 39
— Dr. Gopal K Shyam

Chapter 10. System Architecture 44
— Dr. Gopal K Shyam

Chapter 11. The Network File System 51
— Dr.S,Senthil Kumar

Chapter 12.An Elaboration Of The Process Of Distributed System 56
— Mr.Pakruddin

Chapter 13. Multithreaded Clients 61
— Ramesh Chandra Tripathi

Chapter 14. Client-Side Software for Distribution Transparency 68
— Tushar Mehrotra

Chapter 15. Wide Area Clusters 77
— Gaurav Kumar Rajput

Chapter 16. Migration in Heterogeneous Systems 80
— Aaditya Jain

Chapter 17. An Over View of the Distributed Communication 83
— Ashendra Kumar Saxena

Chapter 18. Introduction to DCE 90
— *Rajendra P. Pandey*

Chapter 19. An Overview of Naming in Distributed System 100
— *Vineet Saxena*

Chapter 20. The Implementation of a Name Space 108
— *Amit Kumar Bishnoi*

Chapter 21. An Introduction of Coordination 118
— *Shambhu Bhardwaj*

Chapter 22. A Ring Algorithm 128
— *Ajay Rastogi*

Chapter 23. An Analysis of Consistency and Replication 133
— *Manish Joshi*

Chapter 24. Implementing Client-Centric Consistency 138
— *Namit Gupta*

Chapter 25. An Introduction of Fault Tolerance 144
— *Pradeep Kumar Shah*

CHAPTER 1

INTRODUCTION OF THE DISTRIBUTED SYSTEM

Dr. A. Jayachandran, Professor,
Department of Computer Science & Engineering,
Presidency University, Bangalore, Karnataka, India.
Email Id- ajayachandran@presidencyuniversity.in

The pace at which computer systems change was, is, and continues to be overwhelming. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. Moreover, for lack of a way to connect them, these computers operated independently from one another. Starting in the mid-1980s, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors. Initially, these were 8-bit machines, but soon 16-bit, 32-bit, and 64-bit CPUs became common. With multicore CPUs, we now are prefacing the challenge of adapting and developing programs to exploit parallelism. In any case, the current generation of machines have the computing power of the mainframes deployed 30 or 40 years ago, but for 1/1000th of the price or less.

The second development was the invention of high-speed computer networks. Local-area networks or LANs allow thousands of machines within abuilding to be connected in such a way that small amounts of information can be transferred in a few microseconds or so. Larger amounts of data can be moved between machines at rates of billions of bits per second (bps). Wide-area networks or WANs allow hundreds of millions of machines all over the earth to be connected at speeds varying from tens of thousands to hundreds of millions bps [1].

Parallel to the development of increasingly powerful and networked ma- chines, we have also been able to witness miniaturization of computer systems with perhaps the smartphone as the most impressive outcome. Packed with sensors, lots of memory, and a powerful CPU, these devices are nothing less than full-fledged computers. Of course, they also have networking capabilities. Along the same lines, so-called plug computers are finding their way to the Market. These small computers, often the size of a power adapter, can be plugged directly into an outlet and offer near-desktop performance.

The result of these technologies is that it is now not only feasible, but easy, to put together a computing system composed of a large number of networked computers, be they large or small. These computers are generally geographically dispersed, for which reason they are usually said to form a distributed system. The size of a distributed system may vary from a handful of devices, to millions of computers. The interconnection network may be wired, wireless, or a combination of both. Moreover, distributed systems areoften highly dynamic, in the sense that computers can join and leave, with the topology and performance of the underlying network almost continuously changing. In this chapter, we provide an initial exploration of distributed systems and their design goals, and follow that up by discussing some well-known types of systems.

Distributed System

Various definitions of distributed systems have been given in the literature, none of them satisfactory, and none of them in agreement with any of the others. For our purposes it is

sufficient to give a loose characterization. A distributed system is a collection of autonomous computing elementsthat appears to its users as a single coherent system. This definition refers to two characteristic features of distributed systems. The first one is that a distributed system is a collection of computing elements each being able to behave independently of each other. A computing element, which we will generally refer to as a node, can be either a hardware device or a software process [2]. A second feature is that users be they people or applications believe they are dealing with a single system. This means that one way or another the autonomous nodes need to collaborate. Howto establish this collaboration lies at the heart of developing distributed systems. Note that we are not making any assumptions concerning the type of nodes. In principle, even within a single system, they could range from high-performance mainframe computers to small devices in sensor networks. Likewise, we make no assumptions concerning the way that nodes are interconnected.

Collection of Autonomous Computing Elements

Modern distributed systems can, and often will, consist of all kinds of nodes,ranging from very big high-performance computers to small plug computersor even smaller devices. A fundamental principle is that nodes can act independently from each other, although it should be obvious that if they ignore each other, then there is no use in putting them into the same distributed system. In practice, nodes are programmed to achieve common goals, whichare realized by exchanging messages with each other. A node reacts to in- coming messages, which are then processed and, in turn, leading to further communication through message passing.

An important observation is that, as a consequence of dealing with inde- pendent nodes, each one will have its own notion of time. In other words, we cannot always assume that there is something like a global clock. This lack of a common reference of time leads to fundamental questions regarding the synchronization and coordination within a distributed system, which we will come to discuss extensively. The fact that we are dealing with a *collection* of nodes implies that we may also need to manage the membership and organization of that collection. In other words, we may need to register which nodes may or may not belong to the system, and also provide each member with a list of nodes it can directly communicate with.

Managing group membership can be exceedingly difficult, if only for reasons of admission control. To explain, we make a distinction between open and closed groups. In an open group, any node is allowed to join the distributed system, effectively meaning that it can send messages to any other node in the system. In contrast, with a closed group, only the members of that group can communicate with each other and a separate mechanism is needed to let a node join or leave the group [3]. It is not difficult to see that admission control can be difficult. First, a mechanism is needed to authenticate a node, and as we shall see in Chap- ter 9, if not properly designed, managing authentication can easily create a scalability bottleneck. Second, each node must, in principle, check if it is indeed communicating with another group member and not, for example, with an intruder aiming to create havoc. Finally, considering that a member can easily communicate with nonmembers, if confidentiality is an issue in the communication within the distributed system, we may be facing trust issues. Concerning the organization of the collection, practice shows that a distributed system is often organized as an overlay network. In this case, a node is typically a software process equipped with a list of otherprocesses it can directly send messages to. It may also be the case that a neigh- bor needs to be first looked up. Message passing is then done through TCP/IPor UDP channels, higher-level facilities may be available as well. There are roughly two types of overlay networks:

Structured overlay: In this case, each node has a well-defined set of neighbors with whom it can communicate. For example, the nodes are organized in a tree or logical ring.

Unstructured overlay: In these overlays, each node has a number of references to randomly select other nodes.

In any case, an overlay network should, in principle, always be connected, meaning that between any two nodes there is always a communication path allowing those nodes to route messages from one to the other. A well-known class of overlays is formed by peer-to-peer (P2P) networks. It is important to realize that the organization of nodes requires special effort and that it is sometimes one of the more intricate parts of distributed-systems management.

Single Coherent System

As mentioned, a distributed system should appear as a single coherent system. In some cases, researchers have even gone so far as to say that there should be a single-system view, meaning that end users should not even notice that they are dealing with the fact that processes, data, and control are dispersed across a computer network. Achieving a single-system view is often asking too much, for which reason, in our definition of a distributed system, we have opted for something weaker, namely that it appears to be coherent. Roughly speaking, a distributed system is coherent if it behaves according to the expectations of its users. More specifically, in a single coherent system the collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

Offering a single coherent view is often challenging enough. For example, it requires that an end user would not be able to tell exactly on which computer a process is currently executing, or even perhaps that part of a task has been spawned off to another process executing somewhere else. Likewise, where data is stored should be of no concern, and neither should it matter that the system may be replicating data to enhance performance. This so called distribution transparency, is an important design goal of distributed systems. In a sense, it is akin to the approach taken in many Unix-like operating systems in which resources are accessed through a unifying file-system interface, effectively hiding the differences between files, storage devices, and main memory, but also networks.

However, striving for a single coherent system introduces an important trade-off. As we cannot ignore the fact that a distributed system consists of multiple, networked nodes, it is inevitable that at any time only a part of the system fails. This means that unexpected behavior in which, for example, some applications may continue to execute successfully while others come to a grinding halt, is a reality that needs to be dealt with. Although partial failures are inherent to any complex system, in distributed systems they are particularly difficult to hide. It lead Turing-Award winner Leslie Lamport, to describe a distributed system one in which the failure of a computer you didn't even know existed can render your own computer unusable [4].

Middleware and Distributed Systems

To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system. This organization is shown in Figure 1, leading to what is known as middleware.

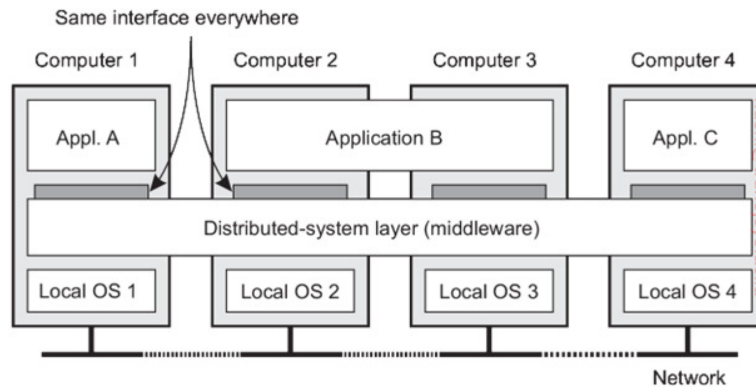

Figure 1: Represented that a Distributed System Organized in a Middleware Layer.

According to the above figure, four networked computers and three applications, of which application B is distributed across computers 2 and 3. Each application is offered the same interface. The distributed system provides the means forcomponents of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonably as possible, the differences in hardware and operating systems from each application.

In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network. Next to resource management, it offers services that can also be found in most operating systems, including:

- **a.** Facilities for inter application communication.
- **b.** Security services.
- **c.** Accounting services.
- **d.** Masking of and recovery from failures.

The main difference with their operating-system equivalents, is that middleware services are offered in a networked environment. Note also that most services are useful to many applications. In this sense, middleware can also be viewed as a container of commonly used components and functions that now no longer have to be implemented by applications separately. To further illustrate these points, let us briefly consider a few examples of typical middleware services.

i. Communication:

A common communication service is the so-called Remote Procedure Call (RPC). An RPC service, allows an application to invoke a function that is implemented and executed on a remote computer as if it was locally available. To thisend, a developer need merely specify the function header expressed in a special programming language, from which the RPC subsystem can then generate the necessary code that establishes remote invocations.

ii. Transactions

Many applications make use of multiple services that are distributed among several computers. Middleware generally offers specialsupport for executing such services in an all-or-nothing fashion, commonly referred to as an atomic transaction. In this case, the applicationdeveloper need only specify the remote services involved, and by following a

standardized protocol, the middleware makes sure that everyservice is invoked, or none at all.

iii. Service Composition

It is becoming increasingly common to develop new applications by taking existing programs and gluing them together. This is notably the case for many Web-based applications, in particular those known as Web services Web-based middleware can help by standardizing the way Web services are accessed and providingthe means to generate their functions in a specific order. A simple example of how service composition is deployed is formed by mashups Web pages that combine and aggregate data from different sources. Well-known mashups are those based on Google maps in which maps are enhanced with extra information such as trip planners or real-timeweather forecasts.

iv. Reliability

As a last example, there has been a wealth of research on pro- viding enhanced functions for building reliable distributed applications. The Horus toolkit allows a developer to build an application as a group of processes such that any message sent by one process is guaranteed to be received by all or no other process. As it turns out, such guarantees can greatly simplify developing distributed applications and are typically implemented as part of the middleware.

v. Design Goals

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. In this section we discuss four important goals that should be met to make building a distributed system worth the effort. Adistributed system should make resources easily accessible; it should hide the fact that resources are distributed across a network; it should be open; and itshould be scalable.

CHAPTER 2

SUPPORTING RESOURCE SHARING

Dr.S.P.Anandaraj, Professor and Hod,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- anandaraj@presidencyuniversity.in

An important goal of a distributed system is to make it easy for users (and applications) to access and share remote resources. Resources can be virtually anything, but typical examples include peripherals, storage facilities, data, files, services, and networks, to name just a few. There are many reasons for wanting to share resources. One obvious reason is that of economics. For example, it is cheaper to have a single high-end reliable storage facility be shared than having to buy and maintain storage for each user separately.

Connecting users and resources also makes it easier to collaborate and exchange information, as is illustrated by the success of the Internet with its simple protocols for exchanging files, mail, documents, audio, and video. The connectivity of the Internet has allowed geographically widely dispersed groups of people to work together by means of all kinds of groupware, that is, software for collaborative editing, teleconferencing, and so on, as is illustrated by multinational software-development companies that have outsourced much of their code production to Asia [5].

However, resource sharing in distributed systems is perhaps best illustrated by the success of file-sharing peer-to-peer networks like Bit Torrent. These distributed systems make it extremely simple for users to share files across the Internet. Peer-to-peer networks are often associated with distribution of media files such as audio and video. In other cases, the technology is used for distributing large amounts of data, as in the case of software updates, backup services, and data synchronization across multiple servers.

Making Distribution Transparent

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers possibly separated by large distances. In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications.

Types of Distribution Transparency

The concept of transparency can be applied to several aspects of a distributed system, of which the most important ones are listed in Table 1. We use the term object to mean either a process or a resource.

Table 1: Represented that the Different forms of Transparency in a Distributed System

Sr. No.	Transparency	Description
1.	Access	Hide differences in data representation and how an object is accessed

2.	Location	Hide where an object is located
3.	Relocation	Hide that an object may be moved to another location while in use
4.	Migration	Hide that an object may move to another location
5.	Replication	Hide that an object is replicated
6.	Concurrency	Hide that an object may be shared by several independent users
7.	Failure	Hide the failure and recovery of an object

Access transparency deals with hiding differences in data representation and the way that objects can be accessed. At a basic level, we want to hide differences in machine architectures, but more important is that we reach agreement on how data is to be represented by different machines and operating systems. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, differences in file operations, or differences in how low-level communication with other processes is to take place, are examples of access issues that should preferably be hidden from users and applications.

An important group of transparency types concerns the location of a pro- cess or resource. Location transparency refers to the fact that users cannot tell where an object is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can often be achieved by assigning only logical names to re- sources, that is, names in which the location of a resource is not secretly encoded. An example of a name is the uniform resource locator (URL), which gives no clue about the actual location of Prentice Hall's main Web server. The URL also gives no clue as to whether the file index.html has always been at its current location or was recently moved there. For example, the entire site may have been moved from one data center to another, yet users should not notice. The latter is an example of relocation transparency, which is becoming increasingly important in the context of cloud computing to which we return later in this chapter.

Where relocation transparency refers to *being* moved by the distributed system, migration transparency is offered by a distributed system when it supports the mobility of processes and resources initiated by users, with- out affecting ongoing communication and operations. A typical example is communication between mobile phones: regardless whether two people are actually moving, mobile phones will allow them to continue their conversation. Other examples that come to mind include online tracking and tracing of goods as they are being transported from one place to another, and teleconferencing (partly) using devices that are equipped with mobile Internet.

As we shall see, replication plays an important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. Replication transparency deals with hiding the fact that several copies of a resource exist, or that several processes are operating in some form of lockstep mode so that one can take over when another fails. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

We already mentioned that an important goal of distributed systems is to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication channels. However, there are also many examples of competitive sharing of resources. For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called concurrency transparency. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. A more refined mechanism is to make use of transactions, but these may be difficult to implement in a distributed system, notably when scalability is an issue.

Last, but certainly not least, it is important that a distributed system provides failure transparency. This means that a user or application does not notice that some piece of the system fails to work properly, and that the system subsequently and automatically recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions. The main difficulty in masking and transparently recovering from failures lies in the inability to distinguish between a dead process and a painfully slowly responding one. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot tell whether the server is actually down or that the network is badly congested.

CHAPTER 3

DEGREE OF DISTRIBUTION TRANSPARENCY

Ms.Bhavya, Assistant Professor,
Department of Computer Science & Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- bhatullabhavya@presidencyuniversity.in

Although distribution transparency is generally considered preferable for any distributed system, there are situations in which attempting to blindly hide all distribution aspects from users is not a good idea. A simple example is requesting your electronic newspaper to appear in your mailbox before 7 AMlocal time, as usual, while you are currently at the other end of the world living in a different time zone. Your morning paper will not be the morning paper you are used to.

Likewise, a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam cannot be expected to hide the fact that Mother Nature will not allow it to send a message from one process to the other in less than approximately 35 milliseconds. Practice shows that it actually takes several hundred milliseconds using a computer network. Signal transmission is not only limited by the speed of light, but also by limited processing capacities and delays in the intermediate switches.

There is also a trade-off between a high degree of transparency and the performance of a system. For example, many Internet applications repeatedly try to contact a server before finally giving up. Consequently, attempting to mask a transient server failure before trying another one may slow down the system as a whole. In such a case, it may have been better to give up earlier,or at least let the user cancel the attempts to make contact.

Another example is where we need to guarantee that several replicas, located on different continents, must be consistent all the time. In other words, if one copy is changed, that change should be propagated to all copies before allowing any other operation. It is clear that a single update operation may now even take seconds to complete, something that cannot be hidden from users.

Finally, there are situations in which it is not at all obvious that hidingdistribution is a good idea. As distributed systems are expanding to devicesthat people carry around and where the very notion of location and context awareness is becoming increasingly important, it may be best to actually expose distribution rather than trying to hide it. An obvious example is making useof location-based services, which can often be found on mobile phones, suchas finding the nearest Chinese take-away or checking whether any of yourfriends are nearby.

There are also other arguments against distribution transparency. Recognizing that full distribution transparency is simply impossible, we should ask ourselves whether it is even wise to pretend that we can achieve it. It maybe much better to make distribution explicit so that the user and application developer are never tricked into believing that there is such a thing as transparency. The result will be that users will much better understand the behavior of a distributed system, and are thus much better prepared to deal with this behavior.

The conclusion is that aiming for distribution transparency may be anice goal when designing and implementing distributed systems, but that it should be considered

together with other issues such as performance and comprehensibility. The price for achieving full transparency may be surprisingly high.

Being Open

Another important goal of distributed systems is openness. An open distributed system is essentially a system that offers components that can easily be used by, or integrated into other systems. At the same time, an open distributed system itself will often consist of components that originate fromelsewhere.

Interoperability, Composability and Extensibility

To be open means that components should adhere to standard rules that describe the syntax and semantics of what those components have to offer (i.e., which service they provide). A general approach is to define services through interfaces using an Interface Definition Language (IDL). Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are given in an informal way by means of natural language.

If properly specified, an interface definition allows an arbitrary process that needs a certain interface, to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate components that operate in exactly the same way.

Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified. However, many interface definitions are not at all complete, so that it is necessary for a developer to add implementation-specific details. Just as important is the fact that specifications do not prescribe what an implementation should look like; they should be neutral.

a. Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.

b. Portability characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system 'B' that implements the same interfaces as 'A'.

Another important goal for an open distributed system is that it should be easy to configure the system out of different components. Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be extensible. For example, in an extensible system, it should be relatively easy to add parts thatrun on a different operating system, or even to replace an entire file system.

Separating Policy from Mechanism

To achieve flexibility in open distributed systems, it is crucial that the system be organized as a collection of relatively small and easily replaceable or adaptable components. This implies that we should provide definitions of not only the highest-level interfaces, that is, those seen by users and applications, but also definitions for interfaces to internal parts of the system and describe how those parts interact. This approach is relatively new. Many

older and even contemporary systems are constructed using a monolithic approach in which components are only logically separated but implemented as one, huge program. This approach makes it hard to replace or adapt a component without affecting the entire system. Monolithic systems thus tend to be closed instead of open.

The need for changing a distributed system is often caused by a component that does not provide the optimal policy for a specific user or application. As an example, consider caching in Web browsers. There are many different parameters that need to be considered:

i. Storage

Where is data to be cached? Typically, there will be an in-memory cache next to storage on disk. In the latter case, the exact position in the local file system needs to be considered.

ii. Exemption

When the cache fills up, which data is to be removed so that newly fetched pages can be stored?

iii. Sharing

Does each browser make use of a private cache, or is a cache to be shared among browsers of different users?

iv. Refreshing

When does a browser check if cached data is still up-to-date? Caches are most effective when a browser can return pages without having to contact the original Web site. However, this bears the risk of returning stale data. Note also that refresh rates are highly dependent on which data is actually cached: whereas timetables for trains hardly change, this is not the case for Web pages showing current highway- traffic conditions, or worse yet, stock prices.

What we need is a separation between policy and mechanism. In the caseof Web caching, for example, a browser should ideally provide facilities foronly storing documents and at the same time allow users to decide whichdocuments are stored and for how long. In practice, this can be implementedby offering a rich set of parameters that the user can set (dynamically). When taking this a step further, a browser may even offer facilities for plugging in policies that a user has implemented as a separate component.

Being Scalable

For many of us, worldwide connectivity through the Internet is as common as being able to send a postcard to anyone anywhere around the world. Moreover, where until recently we were used to having relatively powerful desktop computers for office applications and storage, we are now witnessing that such applications and services are being placed in what has been coined "the cloud," in turn leading to an increase of much smaller networked devicessuch as tablet computers. With this in mind, scalability has become one of the most important design goals for developers of distributed systems.

Scalability Dimensions

Scalability of a system can be measured along at least three different dimensions:

i. **Size Scalability**

A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system withoutany noticeable loss of performance.

ii. **Geographical Scalability**

A geographically scalable system is one in whichthe users and resources may lie far apart, but the fact that communicationdelays may be significant is hardly noticed.

iii. **Administrative Scalability**

An administratively scalable system is one thatcan still be easily managed even if it spans many independent administrative organizations. Let us take a closer look at each of these three scalability dimensions.

Size Scalability

When a system needs to scale, very different types of problems need to be solved. Let us first consider scaling with respect to size.If more users or resources need to be supported, we are often confronted with the limitations of centralized services, although often for very different reasons. For example, many services are centralized in the sense that they are implemented by means of a single server running on a specific machine in the distributed system. In a more modern setting, we may have a group of collaborating servers co-located on a cluster of tightly coupled machines physically placed at the same location. The problem with this scheme is obvious: the server, or group of servers, can simply become a bottleneck when it needs to process an increasing number of requests. To illustrate how this can happen, let us assume that a service is implemented on a single machine.In that case there are essentially three root causes for becoming a bottleneck:

a. The computational capacity, limited by the CPUs

b. The storage capacity, including the I/O transfer rate

c. The network between the user and the centralized service

Let us first consider the computational capacity. Just imagine a service for computing optimal routes taking real-time traffic information into account. It is not difficult to imagine that this may be primarily a compute-bound service requiring several (tens of) seconds to complete a request. If there is only a single machine available, then even a modern high end system will eventually run into problems if the number of requests increases beyond a certain point.

Likewise, but for different reasons, we will run into problems when having a service that is mainly I/O bound. A typical example is a poorly designed centralized search engine. The problem with content-based search queries is that we essentially need to match a query against an entire data set. Even with advanced indexing techniques, we may still face the problem of having to process a huge amount of data exceeding the main-memory capacity of the machine running the service. As a consequence, much of the processing time will be determined by the relatively slow disk accesses and transfer of data between disk and main memory. Simply adding more or higher-speed disks will prove not to be a sustainable solution as the number of requests continues to increase.

Finally, the network between the user and the service may also be the cause of poor scalability. Just imagine a video-on-demand service that needs to stream high-quality video to multiple users. A video stream can easily require a bandwidth of 8 to 10 Mbps, meaning that if a service sets up point-to-point connections with its customers, it may soon hit the limits of the network capacity of its own outgoing transmission lines. There are several solutions to attack size scalability which we discuss below after having looked into geographical and administrative scalability.

Geographical Scalability

Geographical scalability has its own problems. One of the main reasons why it is still difficult to scale existing distributed systems that were designed for local-area networks is that many of them are based on synchronous communication. In this form of communication, a party requesting service, generally referred to as a client, blocks until a reply is sent back from the server implementing the service. More specifically, we often see a communication pattern consisting of many client-server interactions as may be the case with database transactions. This approach generally works fine in LANs where communication between two machines is often at worst a few hundred microseconds. However, in a wide-area system, we need to take into account that interposes communication may be hundreds of milliseconds, three orders of magnitude slower. Building applications using synchronous communication in wide-area systems requires a great deal of care and not just a little patience, notably with a rich interaction pattern between client and server. Another problem that hinders geographical scalability is that communication in wide-area networks is inherently much less reliable than in local-area networks. In addition, we also need to deal with limited bandwidth. The effect is that solutions developed for local-area networks cannot always be easily ported to a wide-area system. A typical example is streaming video. In a home network, even when having only wireless links, ensuring a stable, fast stream of high-quality video frames from a media server to a display is quite simple. Simply placing that same server far away and using a standard TCP connection to the display will surely fail: bandwidth limitations will instantly surface, but also maintaining the same level of reliability can easily cause headaches.

Yet another issue that pops up when components lie far apart is the fact that wide-area systems generally have only very limited facilities for multipoint communication. In contrast, local-area networks often support efficient broadcasting mechanisms. Such mechanisms have proven to be extremely useful for discovering components and services, which is essential from a management point of view. In wide-area systems, we need to develop separate services, such as naming and directory services to which queries can be sent. These support services, in turn, need to be scalable as well and in many cases no obvious solutions exist as we will encounter in later chapters.

Administrative Scalability

Finally, a difficult, and in many cases open, question is how to scale a distributed system across multiple, independent administrative domains. A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security. To illustrate, for many years scientists have been looking for solutions to share their equipment in what is known as a computational grid. In these grids, a global distributed system is constructed as a federation of local distributed systems, allowing a program running on a computer at organization A to directly access resources at organization B.

For example, many components of a distributed system that reside within a single domain can often be trusted by users that operate within that same domain. In such cases, system

administration may have tested and certified applications, and may have taken special measures to ensure that such components cannot be tampered with. In essence, the users trust their system administrators. However, this trust does not expand naturally across domain boundaries.

If a distributed system expands to another domain, two types of security measures need to be taken. First, the distributed system has to protect itself against malicious attacks from the new domain. For example, users from the new domain may have only read access to the file system in its original domain. Likewise, facilities such as expensive image setters or high-performance computers may not be made available to unauthorized users. Second, the new domain has to protect itself against malicious attacks from the distributed system. A typical example is that of downloading programs such as applets in Web browsers. Basically, the new domain does not know what to expect from such foreign code.

As a counterexample of distributed systems spanning multiple administrative domains that apparently do not suffer from administrative scalability problems, consider modern file-sharing peer-to-peer networks. In these cases, end users simply install a program implementing distributed search and download functions and within minutes can start downloading files. Other examples include peer-to-peer applications for telephony over the Internet such as Skype and peer-assisted audio-streaming applications such as spotify What these distributed systems have in common is that end users, and not administrative entities, collaborate to keep the system up and running. At best, underlying administrative organizations such as Internet Service Providers (ISPs) canpolice the network traffic that these peer-to-peer systems cause, but so far such efforts have not been very effective.

CHAPTER 4

SCALING TECHNIQUES

Dr. Pallavi R., Associate Professor,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- pallavi.r@presidencyuniversity.in

Having discussed some of the scalability problems brings us to the question of how those problems can generally be solved. In most cases, scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network. Simply improving their capacity (e.g., by increasing memory, upgrading CPUs, or replacing network modules) is often a solution, referred to as scaling up. When it comes to scaling out, that is, expanding the distributed system by essentially deploying more machines, there are basically only three techniques we can apply: hiding communication latencies, distribution of work, and replication.

Hiding Communication Latencies

Hiding communication latencies is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote-service requests as much as possible. For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work atthe requester's side. Essentially, this means constructing the requesting application in such a way that it uses only asynchronous communication. Whena reply comes in, the application is interrupted and a special handler is calledto complete the previously issued request. Asynchronous communication can often be used in batch-processing systems and parallel applications in which independent tasks can be scheduled for execution while another task is waiting for communication to complete. Alternatively, a new thread of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue.

However, there are many applications that cannot make effective use of asynchronous communication. For example, in interactive applications when a user sends a request he will generally have nothing better to do than towait for the answer. In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service. A typical case where this approach works is accessing databases using forms.Filling in forms can be done by sending a separate message for each field and waiting for an acknowledgment from the server, as shown in Figure 2(a). For example, the server may check for syntactic errors before accepting an entry.A much better solution is to ship the code for filling in the form, and possibly checking the entries, to the client, and have the client return a completed form, as shown in Figure 2(b). This approach of shipping code is widely supported by the Web by means of Java applets and JavaScript.

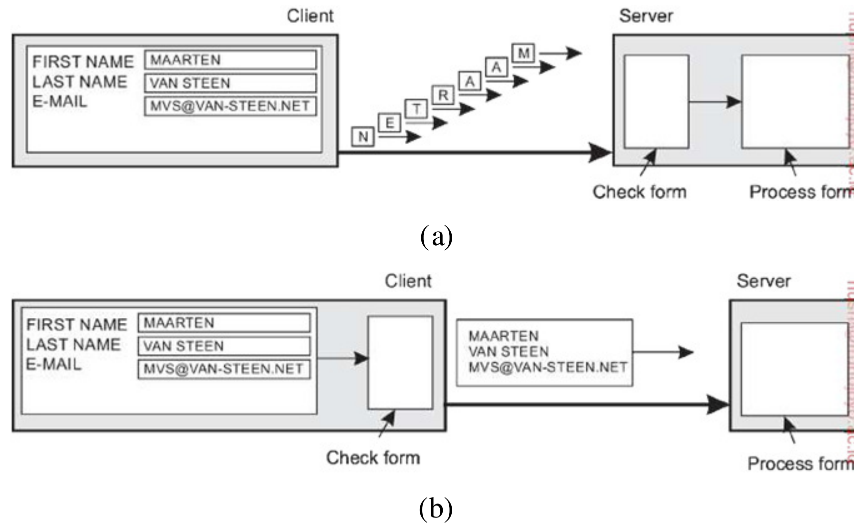

Figure 2: The difference between letting (a) a server or (b) a client checkforms as they are being filled.

Partitioning and Distribution

Another important scaling technique is partitioning and distribution, which involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system. A good example of partitioning and distribution is the Internet Domain Name System (DNS). The DNS name space is hierarchically organized into a tree of domains, which are divided into no overlapping zones, as shown for the original DNS. The names in each zone are handled by a single name server. Without going into too many details now we return to DNS extensively one can think of each path name being the name of a host in the Internet, and is thus associated with a network address of that host. Basically, resolving a name means returning the network address of the associated host. Consider, for example, the name flits.cs.vu.nl. To resolve this name, it is first passed to the server of zone Z1 as display in Figure 3, which returns the address of the server for zone Z2, to which the rest of name, flits.cs.vu, can be handed. The server for Z2 will return the address of the server for zone Z3, which is capable of handling the last part of the name and will return the address of the associated host.

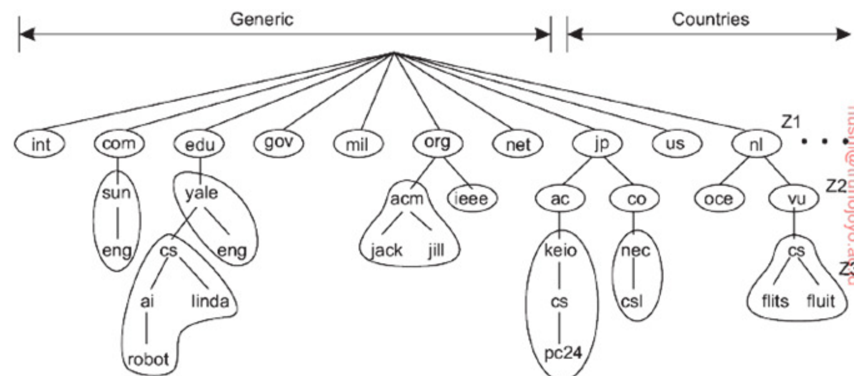

Figure 3: Display the Example of Dividing the DNS Name Space into Zones.

This examples illustrates how the naming service, as provided by DNS, is distributed across several machines, thus avoiding that a single server has to deal with all requests for name resolution. As another example, consider the World Wide Web. To most users, the

Web appears to be an enormous document-based information system in which each document has its own unique name in the form of a URL. Conceptually, it may even appear as if there is only a single server. However, the Web is physically partitioned and distributed across a few hundred million servers, each handling a number of Web documents. The name of the server handling a document is encoded into that document's URL. It is only because of this distribution of documents that the Web has been capable of scaling to its current size.

Replication

Considering that scalability problems often appear in the formof performance degradation, it is generally a good idea to actually replicate components across a distributed system. Replication not only increases availability, but also helps to balance the load between components leading to better performance. Also, in geographically widely dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before. Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. However, in contrast to replication, caching is a decision made by the client of a resource and not by the owner of a resource.

There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to consistency problems. To what extent inconsistencies can be tolerated depends highly on the usage of a resource. For example, many Web users find it acceptable that their browser returns a cached document of which the validity has not been checked for the last few minutes. However, there are also many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchanges and auctions. The problem with strong consistency is that an update must be immediately propagated to all other copies. Moreover, if two updates happen concurrently, it is often also required that updates are processed in the same order everywhere, introducing an additional global ordering problem.

Replication therefore often requires some global synchronization mechanism. Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way, if alone because network latencies have a natural lower bound. Consequently, scaling by replication may introduce other, inherently nonsalable solutions. When considering these scaling techniques, one could argue that size scalability is the least problematic from a technical point of view. In many cases, increasing the capacity of a machine will save the day, although perhaps there is a high monetary cost to pay. Geographical scalability is a much tougher problem as network latencies are naturally bound from below.

As a consequence, we may be forced to copy data to locations close to where clients are, leading to problems of maintaining copies consistent. Practice shows that combining distribution, replication, and caching techniques with different forms of consistency generally leads to acceptable solutions. Finally, administrative scalability seems to be the most difficult problem to solve, partly because we need to deal with nontechnical issues, such as politics of organizations and human collaboration. The introduction and now widespread use of peer-to-peer technology has successfully demonstrated what can be achieved if end users are put in control However, peer-to-peer networks are obviously not the universal solution to all administrative scalability problems.

Pitfalls

It should be clear by now that developing a distributed system is a formidable task. As we will see many times throughout this book, there are so many issues to consider at the same time that it seems that only complexity can be the result. Nevertheless, by following a number of design principles, distributed systems can be developed that strongly adhere to the goals we set out in this chapter.

Distributed systems differ from traditional software because components are dispersed across a network. Not taking this dispersion into account during design time is what makes so many systems needlessly complex and results in flaws that need to be patched later on. Peter Deutsch, at the time working at Sun Microsystems, formulated these flaws as the following false assumptions that everyone makes when developing a distributed application for the first time:

- **a.** The network is reliable
- **b.** The network is secure
- **c.** The network is homogeneous
- **d.** The topology does not change
- **e.** Latency is zero
- **f.** Bandwidth is infinite
- **g.** Transport cost is zero
- **h.** There is one administrator

Note how these assumptions relate to properties that are unique to distributed systems: reliability, security, heterogeneity, and topology of the network; latency and bandwidth; transport costs; and finally administrative domains. When developing no distributed applications, most of these issues will most likely not show up. Most of the principles we discuss in this book relate immediately to these assumptions. In all cases, we will be discussing solutions to problems that are caused by the fact that one or more assumptions are false. For example, reliable networks simply do not exist and lead to the impossibility of achieving failure transparency. We devote an entire chapter to deal with the fact that networked communication is inherently insecure. We have already argued that distributed systems need to be open and take heterogeneity into account. Likewise, when discussing replication for solving scalability problems, we are essentially tackling latency and bandwidth problems. We will also touch upon management issues at various points throughout this book.

Types of Distributed Systems

Before starting to discuss the principles of distributed systems, let us first take a closer look at the various types of distributed systems. We make a distinction between distributed computing systems, distributed information systems, and pervasive systems which are naturally distributed.

High Performance Distributed Computing

An important class of distributed systems is the one used for high-performance computing tasks. Roughly speaking, one can make a distinction between two subgroups. In cluster computing the underlying hardware consists of a collection of similar workstations or PCs,

closely connected by means of a high-speed local-area network. In addition, each node runs the same operating system. The situation becomes very different in the case of grid computing. This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology. From the perspective of grid computing, a next logical step is to simply outsource the entire infrastructure that is needed for compute-intensive applications. In essence, this is what cloud computing is all about: providing the facilities to dynamically construct an infrastructure and compose whatis needed from available services. Unlike grid computing, which is strongly associated with high-performance computing, cloud computing is much more than just providing lots of resources. We discuss it briefly here, but will return to various aspects throughout the book.

Cluster Computing

Cluster computing systems became popular when the price or performance ratio of personal computers and workstations improved. At a certain point, itbecame financially and technically attractive to build a supercomputer usingoff-the-shelf technology by simply hooking up a collection of relatively simple computers in a high-speed network. In virtually all cases, cluster computing is used for parallel programming in which a single compute intensive programis run in parallel on multiple machines. One widely applied example of a cluster computer is formed by Linux- based Beowulf clusters, of which the general configuration. Each cluster consists of a collection of compute nodes that are con- trolled and accessed by means of a single master node. The master typically handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system. As such, the master actually runs the middleware needed for theexecution of programs and management of the cluster, while the compute nodes are equipped with a standard operating system extended with typical middleware functions for communication, storage, fault tolerance, and so on. Apart from the master node, the compute nodes are thus seen to be highly identical.

An even more symmetric approach is followed in the MOSIX system MOSIX attempts to provide a single-system image of a cluster, meaning that to a process a cluster computer offers the ultimate distribution transparency by appearing to be a single computer. As we mentioned, pro- viding such an image under all circumstances is impossible. In the case of MOSIX, the high degree of transparency is provided by allowing processes to dynamically and preemptively migrate between the nodes that make up the cluster. Process migration allows a user to start an application on any node, after which it can transparently move toother nodes, for example, to make efficient use of resources.

However, several modern cluster computers have been moving away from these symmetric architectures to more hybrid solutions in which the middleware is functionally partitioned across different nodes. The advantage of such a separation is obvious having compute nodes with dedicated, lightweight operating systems will most likely provide optimal performance for compute-intensive applications. Likewise, storage functionality can most likely be optimally handled by other speciallyconfigured nodes such as file and directory servers. The same holds for otherdedicated middleware services, including job management, database services, and perhaps general Internet access to external services.

Grid Computing

A characteristic feature of traditional cluster computing is its homogeneity. In most cases, the computers in a cluster are largely the same, have the same operating system, and are

all connected through the same network. However, as we just discussed, there has been a trend towards more hybrid architectures in which nodes are specifically configured for certain tasks. This diversity is even more prevalent in grid computing systems: no assumptions are made concerning similarity of hardware, operating systems, networks, administrative domains, security policies, etc. A key issue in a grid-computing system is that resources from different organizations are brought together to allow the collaboration of a group of people from different institutions, indeed forming a federation of systems. Such a collaboration is realized in the form of a virtual organization. The processes belonging to the same virtual organization have access rights to the resources that are provided to that organization. Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases. In addition, special networked devices such as telescopes, sensors, etc., can be provided as well. Given its nature, much of the software for realizing grid computing evolves around providing access to resources from different administrative domains,and to only those users and applications that belong to a specific virtual organization. For this reason, focus is often on architectural issues.

The architecture consists of four layers. The lowest fabric layer provides interfaces to local resources at a specific site. Note that these interfaces aretailored to allow sharing of resources within a virtual organization. Typically,they will provide functions for querying the state and capabilities of a resource,along with functions for actual resource management. The connectivity layer consists of communication protocols for supporting grid transactions that span the usage of multiple resources. For example,protocols are needed to transfer data between resources, or to simply accessa resource from a remote location. In addition, the connectivity layer will contain security protocols to authenticate users and resources. Note that in many cases human users are not authenticated; instead, programs acting onbehalf of the users are authenticated. In this sense, delegating rights from a user to programs is an important function that needs to be supported in the connectivity layer.

The resource layer is responsible for managing a single resource. It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer. For example, this layer will offer functions for obtaining configuration information on a specific resource, or, in general,to perform specific operations such as creating a process or reading data. The resource layer is thus seen to be responsible for access control, and hence will rely on the authentication performed as part of the connectivity layer. The next layer in the hierarchy is the collective layer. It deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on. Unlike the connectivity and resource layer, each consisting of a relatively small, standard collection of protocols, the collective layer may consist of many different protocols reflecting the broad spectrum of services it may offer to a virtual organization.

Finally, the application layer consists of the applications that operate within a virtual organization and which make use of the grid computing environment. Typically the collective, connectivity, and resource layer form the heart of what could be called a grid middleware layer. These layers jointly provide access to and management of resources that are potentially dispersed acrossmultiple sites. An important observation from a middleware perspective is that in grid computing the notion of a site or administrative unit is common. This prevalence is emphasized by the gradual shift toward a service-oriented architecture in which sites offer access to the various layers through a collection of Web services. This, by now,

has led to the definition of an alternative architecture known as the Open Grid Services Architecture (OGSA). OGSA is based upon the original ideas as formulated yet having gone through a standardization process makes it complex, to say the least. OGSA implementations generally follow Web service standards.

CHAPTER 5

AN ELABORATION OF THE CLOUD COMPUTING

Dr. Blessed Prince, Associate Professor,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- blessedprince@presidencyuniversity.in

While researchers were pondering on how to organize computational grids that were easily accessible, organizations in charge of running data centers were facing the problem of opening up their resources to customers. Eventually, this lead to the concept of utility computing by which a customer could upload tasks to a data center and be charged on a per-resource basis. Utility computing formed the basis for what is now called cloud computing.

Cloud computing is characterized by an easily usable and accessible pool of virtualized resources. Which and how resources are used can be configured dynamically, providing the basis for scalability: if more work needs to be done, a customer can simply acquire more resources. The link to utility computing is formed by the fact that cloud computing is generally based on a pay-per-use model in which guarantees are offered by means of customized service level agreements (SLAs).

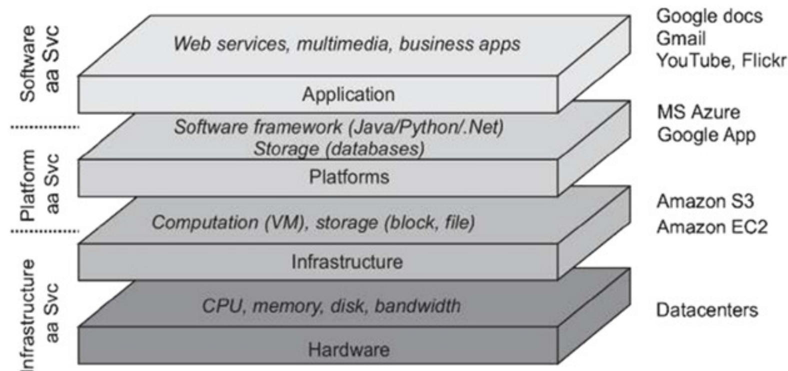

Figure 1: Represented that the Organization of Clouds

In practice, clouds are organized into four layers, as shown in Figure 1.

i. Hardware

The lowest layer is formed by the means to manage the necessary hardware: processors, routers, but also power and cooling systems. It is generally implemented at data centers and contains the resources that customers normally never get to see directly.

ii. Infrastructure

This is an important layer forming the backbone for most cloud computing platforms. It deploys virtualization techniques to provide customers an infrastructure consisting of virtual storage and computing resources. Indeed, nothing is what it seems: cloud computing evolves around allocating and managing virtual storage devices and virtual servers.

iii. Platform

One could argue that the platform layer provides to a cloud- computing customer what an operating system provides to application developers, namely the means to easily develop and deploy applications that need to run in a cloud. In practice, an application developer is offered a vendor-specific API, which includes calls to uploading and executing a program in that vendor's cloud. In a sense, this is comparable the Unix exec family of system calls, which take an executable file as parameter and pass it to the operating system to be executed. Also like operating systems, the platform layer provides higher-level abstractions for storage and such. For example, as we discuss in more detail later, the Amazon S3 storage system is offered to the application developer in the form of an API allowing locally created files to be organized and stored in buckets. A bucket is somewhat comparable to a directory. By storing a file in a bucket, that file is automatically uploaded to the Amazon cloud[6].

iv. Application

Actual applications run in this layer and are offered to users for further customization. Well-known examples include those found in office suites (text processors, spreadsheet applications, presentation applications, and so on). It is important to realize that these applications are again executed in the vendor's cloud. As before, they can be compared to the traditional suite of applications that are shipped when installing an operating system.

Cloud-computing providers offer these layers to their customers through various interfaces including command-line tools, programming interfaces, and Web interfaces, leading to three different types of services:

a. **Infrastructure-as-a-Service** (**IaaS**) covering the hardware and infrastructure layer

b. **Platform-as-a-Service** (**PaaS**) covering the platform layer

c. **Software-as-a-Service** (**SaaS**) in which their applications are covered

As of now, making use of clouds is relatively easy, and we discuss in later chapters more concrete examples of interfaces to cloud providers. As a consequence, cloud computing as a means for outsourcing local computing infrastructures has become a serious option for many enterprises. However, there are still a number of serious obstacles including provider lock-in, security and privacy issues, and dependency on the availability of services, to mention a few Also, because the details on how specific cloud computations are actually carried out are generally hidden, and even perhaps unknown or unpredictable, meeting performance demands may be impossible to arrange in advance. Cloud computing is no longer a hype, and certainly a serious alternative to maintaining huge local infrastructures, yet there is still a lot of room for improvement.

Distributed Information Systems

Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience. Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an enterprise-wide information system. We can distinguish several levels at which integration can take place. In many cases, a networked application simply consists of a server running that application often including a database and making it available to remote programs, called clients. Such clients send a request to the server for executing a specific operation, after which a response is sent back. Integration at the lowest level allows clients to wrap a number of

requests, possibly for different servers, into a single larger request and have it executed as a distributedtransaction. The key idea is that all, or none of the requests are executed. As applications became more sophisticated and were gradually separated into independent components notably distinguishing database components from processing components, it became clear that integration should also take place by letting applications communicate directly with each other. This has now lead to a huge industry that concentrates on Enterprise Application Integration (EAI).

CHAPTER 6

DISTRIBUTED TRANSACTION PROCESSING

Dr. Jayavadivel, Associate Professor,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- jayavadivel.ravi@presidencyuniversity.in

To clarify our discussion, we concentrate on database applications. In practice, operations on a database are carried out in the form of transactions. Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system. The exact list of primitives depends on what kinds of objects are being used in the transaction. In a mail system, there might be primitives to send, receive, and forward mail.In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on,are also allowed inside a transaction. In particular, remote procedure calls (RPCs), that is, procedure calls as mention in Table 1, to remote servers, are often also encapsulatedin a transaction, leading to what is known as a transactional RPC.

Table 1: Represented that the Example Primitives for Transactions.

Sr. No.	Primitive	Description
	BEGIN_TRANSACTION	Mark the start of a transaction
	END_TRANSACTION	Terminate the transaction and try to commit
	ABORT_TRANSACTION	Kill the transaction and restore the old values
	READ	Read data from a file, a table, or otherwise
	WRITE	Write data to a file, a table, or otherwise

BEGIN_TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. The characteristic feature of a transaction is either all of these operations are executed or none are executed. These may be system calls,library procedures, or bracketing statements in a language, depending on the implementation. This all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions adhere to the so-called ACID properties:

a. **Atomic**: To the outside world, the transaction happens indivisibly
b. **Consistent**: The transaction does not violate system invariants
c. **Isolated**: Concurrent transactions do not interfere with each other
d. **Durable**: Once a transaction commits, the changes are permanent

In distributed systems, transactions are often constructed as a number of sub transactions, jointly forming a nested transaction the top-level transaction may fork off children that run in

parallel with one another, on different machines, to gain performance or simplify programming. Each of these children may also execute one or more sub transactions, or fork off its own children. Sub transactions give rise to a subtle, but important, problem. Imagine that a transaction starts several sub transactions in parallel, and one of these commits, making its results visible to the parent transaction. After further computation, the parent aborts, restoring the entire system to the state it had before the top-level transaction started. Consequently, the results of the sub transaction that committed must nevertheless be undone. Thus the permanence referred to above applies only to top-level transactions[7].

Since transactions can be nested arbitrarily deep, considerable administration is needed to get everything right. The semantics are clear, however. When any transaction or sub-transaction starts, it is conceptually given a private copy of all data in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent's universe. Thus if a sub transaction commits and then later a new sub transaction is started, the second one sees the results produced by the first one. Likewise, if an enclosing transaction aborts, all its underlying sub transactions have to be aborted as well. And if several transactions are started concurrently, the result is as if they ran sequentially in some unspecified order.

Nested transactions are important in distributed systems, for they provide a natural way of distributing a transaction across multiple machines. They follow a logical division of the work of the original transaction. For example, a transaction for planning a trip by which three different flights need to be reserved can be logically split up into three sub transactions. Each of these sub transactions can be managed separately and independently of the other two.

In the early days of enterprise middleware systems, the component that handled distributed (or nested) transactions formed the core for integrating applications at the server or database level. This component was called a transaction processing monitor or TP monitor for short. Its main task was to allow an application to access multiple server or databases by offering it a transactional programming model.

An important observation is that applications wanting to coordinate several sub transactions into a single transaction did not have to implement this coordination themselves. By simply making use of a TP monitor, this coordination was done for them. This is exactly where middleware comes into play: it implements services that are useful for many applications avoiding that such services have to be implemented over and over again by application developers.

Enterprise Application Integration

As mentioned, the more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to integrate applications independently from their databases. In particular, ap- plication components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems. This need for interapplication communication led to many different communication models, the main idea was that existing applications could directly exchange information.

Several types of communication middleware exist. With remote procedure calls (RPC), an application component can effectively send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the caller. Likewise, theresult will be sent back and returned to the application as the result

of the procedure call. As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as remote method invocations (RMI). An RMI is essentially the same as an RPC, except that it operates on objects instead of functions.

RPC and RMI have the disadvantage that the caller and callee both need to be up and running at the time of communication. In addition, they need to know exactly how to refer to each other. This tight coupling is often experienced as a serious drawback, and has led to what is known as message- oriented middleware, or simply MOM. In this case, applications send messages to logical contact points, often described by means of a subject. Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications. These so-called publish/subscribe systems form an important and expanding class of distributed systems [8].

Pervasive Systems

The distributed systems discussed so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network. To a certain extent, this stability is realized through the various techniques for achieving distribution transparency. For example, there are many ways how we can create the illusion that only occasionally components may fail. Likewise, there are all kinds of means to hide the actualnetwork location of a node, effectively allowing users and applications to believe that nodes stay put.

However, matters have changed since the introduction of mobile and embedded computing devices, leading to what are generally referred to as pervasive systems. As its name suggests, pervasive systems are intended to naturally blend into our environment. What makes them unique in comparison to the computing and information systems described so far, is that the separation between users and system components is much more blurred. There is often no single dedicated interface, such as a screen/keyboard combination. Instead, a pervasive system is often equipped with many sensors that pick up various aspects of a user's behavior. Likewise, it may have a myriad of actuators to provide information and feedback, often even purposefully aiming to steer behavior.

Many devices in pervasive systems are characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices. These are not necessarily restrictive characteristics, as is illustrated by smartphones and their role in what is now coined as the Internet of Internet. Nevertheless, notably the fact that we often need to deal with the intricacies of wireless and mobile communication, will require special solutions to make a pervasive system as transparent or unobtrusive as possible. In the following, we make a distinction between three different types of pervasive systems, although there is considerable overlap between the threetypes: ubiquitous computing systems, mobile systems, and sensor networks. This distinction allows us to focus on different aspects of pervasive systems.

Ubiquitous computing systems

So far, we have been talking about pervasive systems to emphasize that its elements have spread through in many parts of our environment. In a ubiquitous computing system we go one step further: the system is pervasive and continuously present.

The latter means that a user will be continuously interacting with the system, often not even being aware that interaction is taking place. Poslad [2009] describes the core requirements for a ubiquitous computing system roughly as follows:

a. Distribution: Devices are networked, distributed, and accessible in a transparent manner.

b. Interaction: Interaction between users and devices is highly unobtrusive.

c. Context awareness: The system is aware of a user's context in order to optimize interaction

d. Autonomy: Devices operate autonomously without human intervention, and are thus highly self-managed

e. Intelligence: The system as a whole can handle a wide range of dynamic actions and interactions

Let us briefly consider these requirements from a distributed-systems perspective:

Distribution

As mentioned, a ubiquitous computing system is an example of a distributed system: the devices and other computers forming the nodes of a system are simply networked and work together to form the illusion of a single coherent system. Distribution also comes naturally: there will be devices close to users such as sensors and actuators, connected to computers hidden from view and perhaps even operating remotely in a cloud. Most, if not all, of the requirements regarding distribution transparency should therefore hold.

Interaction

When it comes to interaction with users, ubiquitous computing systems differ a lot in comparison to the systems we have been discussing so far. End users play a prominent role in the design of ubiquitous systems, meaning that special attention needs to be paid to how the interact between users and core system takes place. For ubiquitous computing systems, much of the interaction by humans will be implicit, with an implicit action being defined as one "that is not primarily aimed to interact with a computerized system but which such a system understands as input". In other words, a user could be mostly unaware of the fact that input is being provided to a computer system. From a certain perspective, ubiquitous computing can be said to seemingly hide interfaces.

A simple example is where the settings of a car's driver's seat, steering wheel, and mirrors is fully personalized. If Bob takes a seat, the system will recognize that it is dealing with Bob and subsequently makes the appropriate adjustments. The same happens when Alice uses the car, while an unknown user will be steered toward making his or her own adjustments. This example already illustrates an important role of sensors in ubiquitous computing, namely as input devices that are used to identify a situation a specific person apparently wanting to drive, whose input analysis leads to actions making adjustments. In turn, the actions may lead to natural reactions, for example that Bob slightly changes the seat settings. The system will have to take all implicit and explicit actions by the user into account and react accordingly.

CHAPTER 7

CONTEXT AWARENESS

Dr. Gopal K Shyam, Professor and HoD,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- gopalkirshna.shyam@presidencyuniversity.in

Reacting to the sensory input, but also the explicit input from users is more easily said than done. What a ubiquitous computing system needs to do, is to take the context in which interactions take place into account. Context awareness also differentiates ubiquitous computing systems from the more traditional systems we have been discussing before as “any information that can be used to characterize the situation of entities that is whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves.” In practice, context is often characterized by location, identity, time, and activity: the where, who, when, and what. A system will need to have the necessary (sensory) input to determine one or several of these context types.

Autonomy

An important aspect of most ubiquitous computing systems is that explicit systems management has been reduced to a minimum. In a ubiquitous computing environment there is simply no room for a systems administrator to keep everything up and running. As a consequence, the system as a whole should be able to act autonomously, and automatically react to changes. This requires a myriad of techniques of which several will be discussed throughout this book. To give a few simple examples, think of the following:

i. Address Allocation

In order for networked devices to communicate, they need an IP address. Addresses can be allocated automatically using protocols like the Dynamic Host Configuration Protocol (DHCP).

ii. Adding Devices

It should be easy to add devices to an existing system. A step towards automatic configuration is realized by the Universal Plug and Play Protocol (UPnP). Using UPnP, devices can discover each other and make sure that they can set up communication channels between them.

iii. Automatic Updates

Many devices in a ubiquitous computing system should be able to regularly check through the Internet if their software should be updated. If so, they can download new versions of their components and ideally continue where they left off. Admittedly, these are very simple examples, but the picture should be clear that manual intervention is to be kept to a minimum. We will be discussing many techniques related to self-management in detail throughout the book.

iv. Intelligence

Finally, it mentions that ubiquitous computing systems often use methods and techniques from the field of artificial intelligence. What this means, is that in many cases a wide range of advanced algorithms and models need to be deployed to handle incomplete input, quickly react to a changing environment, handle unexpected events, and so on. The extent to which this can or should be done in a distributed fashion is crucial from the perspective of distributed systems. Unfortunately,distributed solutions for many problems in the field of artificial intelligence are yet to be found, meaning that there may be a natural tension between the first requirement of networked and distributed devices, and advanced distributed information processing.

Mobile Computing Systems

As mentioned, mobility often forms an important component of pervasive systems, and many, if not all aspects that we have just discussed also apply to mobile computing. There are several issues that set mobile computing aside to pervasive systems in general. First, the devices that form part of a mobile system may vary widely. Typically, mobile computing is now done with devices suchas smartphones and tablet computers. However, completely different types of devices are now using the Internet Protocol (IP) to communicate, placing mobile computing in a different perspective. Such devices include remote controls, pagers, active badges, car equipment, various GPS-enabled devices, and so on. A characteristic feature of all these devices is that they use wirelesscommunication. Mobile implies wireless so it seems although there are exceptions to the rules.

Second, in mobile computing the location of a device is assumed to change over time. A changing location has its effects on many issues. For example, if the location of a device changes regularly, so will perhaps the services that are locally available. As a consequence, we may need to pay special attentionto dynamically discovering services, but also letting services announce their presence. In a similar vein, we often also want to know where a device actually is. This may mean that we need to know the actual geographical coordinates of a device such as in tracking and tracing applications, but it may also require that we are able to simply detect its network position.

Changing locations also has a profound effect on communication. To illustrate, consider a (wireless) mobile ad hoc network, generally abbreviated as a MANET. Suppose that two devices in a MANET have discovered each other in the sense that they know each other's network address. How do we route messages between the two? Static routes are generally not sustainable as nodes along the routing path can easily move out of their neighbor's range,invalidating the path. For large MANETs, using a priori set-up paths is nota viable option. What we are dealing with here are so-called disruption- tolerant networks: networks in which connectivity between two nodes can simply not be guaranteed. Getting a message from one node to another may then be problematic, to say the least [9].

The trick in such cases, is not to attempt to set up a communication path from the source to the destination, but to rely on two principles. Obviously, any type of flooding will impose redundantcommunication, but this may be the price we have to pay. Second, in a disruption-tolerant network, we let an intermediate node store a received message until it encounters another node to which it can pass it on. In other words, a node becomes a temporary carrier of a message, as sketched in Figure 2. Eventually, the message should reach its destination. It is not difficult to imagine that selectively passing messages to encountered nodes may help to ensure efficient delivery. For example, if nodes are known to belong to a certain class, and the source and destination belong tothe same class, we may

decide to pass messages only among nodes in thatclass. Likewise, it may prove efficient to pass messages only to well-connected nodes, that is, nodes who have been in range of many other nodes in the recent past.

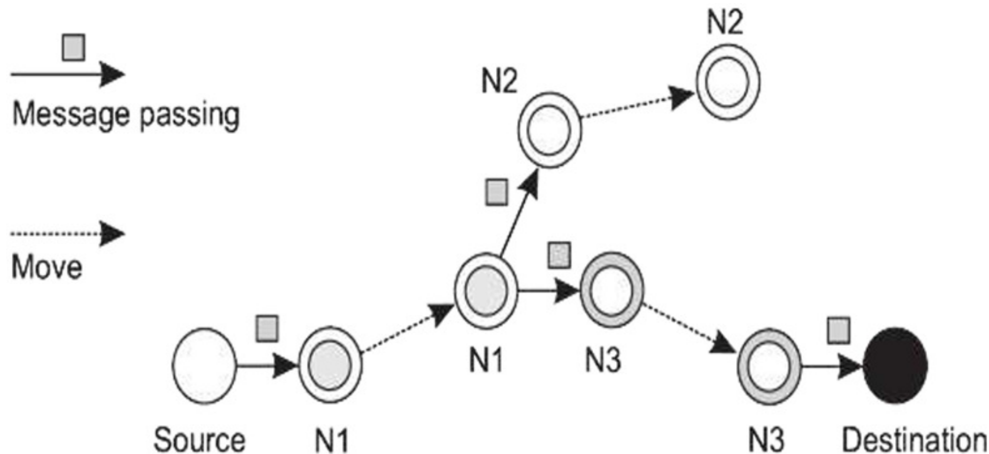

Figure 2: Represented that the Passing messages in a Disruption Tolerant Network.

Sensor Networks

Our last example of pervasive systems is sensor networks. These networks in many cases form part of the enabling technology for pervasiveness and we see that many solutions for sensor networks return in pervasive applications. What makes sensor networks interesting from a distributed system's perspective is that they are more than just a collection of input devices. Instead, as we shall see, sensor nodes often collaborate to efficiently process the sensed data in an application-specific manner, making them very different from, for example, and traditional computer networks.

A sensor network generally consists of tens to hundreds or thousands of relatively small nodes, each equipped with one or more sensing devices. In addition, nodes can often act as actuators a typical example being the automatic activation of sprinklers when a fire has been detected. Many sensor networks use wireless communication, and the nodes are often battery powered. Their limited resources, restricted communication capabilities, and constrained power consumption demand that efficiency is high on the list of design criteria. When zooming into an individual node, we see that, conceptually, theydo not differ a lot from "normal" computers: above the hardware there is a software layer akin to what traditional operating systems offer, including low level network access, access to sensors and actuators, memory management, and so on. Normally, support for specific services is included, such as localization, local storage think of additional flash devices, and convenient communication facilities such as messaging and routing. However, similar to other networked computer systems, additional support is needed to effectively deploy sensor network applications. In distributed systems, this takes the form of middleware.

One typical aspect in programming support is the scope provided by communication primitives. This scope can vary between addressing the physical neighborhood of a node, and providing primitives for system wide communication. In addition, it may also be possible to address a specific group of nodes. Likewise, computations may be restricted to an individual node, a group of nodes, or affect all nodes. To illustrate, use so-called abstract regions allowing a node to identify a neighborhood from where it can, for example, gather information:

```
Line 1.    region  = k_nearest_region.create(8);
Line 2.    reading = get_sensor_reading();
Line 3.    region.putvar(reading_key, reading);
Line 4.    max_id  = region.reduce(OP_MAXID, reading_key);
```

In line 1, a node first creates a region of its eight nearest neighbors, after which it fetches a value from its sensor(s). This reading is subsequently written to the previously defined region to be defined using the key reading_key. In line 4, the node checks whose sensor reading in the defined region was the largest, which is returned in the variable max_id.

As another related example, consider a sensor network as implementing a distributed database, which is, according to one of four possible ways of accessing data. This database view is quite common and easy to understand when realizing that many sensor networks are deployed for measurement and surveillance applications. In these cases, an operator would like to extract information from a part of the network by simply issuing queries such as "What is the northbound traffic load on highway 1 as Santa Cruz?" Such queries resemble those of traditional databases. In this case, the answer will probably need to be provided through collaboration of many sensors along highway, while leaving other sensors untouched.

To organize a sensor network as a distributed database, there are essentially two extremes, sensors do not cooperate but simply send their data to a centralized database located at the operator's site. The other extreme is to forward queries to relevant sensors and to let each compute an answer, requiring the operator to aggregate the responses.

Neither of these solutions is very attractive. The first one requires that sensors send all their measured data through the network, which may waste network resources and energy. The second solution may also be wasteful as it discards the aggregation capabilities of sensors which would allow much less data to be returned to the operator.

What is needed are facilities for in-network data processing, similar to the previous example of abstract regions. In-network processing can be done in numerous ways. One obvious one is to forward a query to all sensor nodes along a tree encompassing all nodes and to subsequently aggregate the results as they are propagated back to the root, where the initiator is located. Aggregation will take place where two or more branches of the tree come together. As simple as this scheme may sound, it introduces difficult questions:

- How do we (dynamically) set up an efficient tree in a sensor network?
- How does aggregation of results take place? Can it be controlled?
- What happens when network links fail?

These questions have been partly addressed in Tiny-DB, which implements a declarative interface to wireless sensor networks. In essence, Tiny-DB can use any tree-based routing algorithm. An intermediate node will collect and aggregate the results from its children, along with its own findings, and send that toward the root. To make matters efficient, queries span a period of time allowing for careful scheduling of operations so that network resources and energy are optimally consumed[10].

However, when queries can be initiated from different points in the network, using single-rooted trees such as in Tiny-DB may not be efficient enough. As an alternative, sensor networks may be equipped with special nodes where results are forwarded to, as well as the queries related to those results. To give a simple example, queries and results related to temperature readings maybe collected at a different location than those related to humidity measurements. This approach corresponds directly to the notion of publish or subscribe systems.

CHAPTER 8

ARCHITECTURE OF DISTRIBUTED SYSTEM

S.poornima, Assistant Professor,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- poornima.s@presidencyuniversity.in

Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines. To master their complexity, it is crucial that these systems are properly organized. There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between, on the one hand, the logical organization of the collection of software components, and on the other hand the actual physical realization. The organization of distributed systems is mostly about the software components that constitute the system. These software architectures tell us how the various software components are to be organized and how they should interact. In this chapter we will first pay attention to some commonly applied architectural styles toward organizing computer systems [11]. An important goal of distributed systems is to separate applications from underlying platforms by providing a middleware layer. Adopting such a layer is an important architectural decision, and its main purpose is to provide distribution transparency. However, trade-offs need to be made to achieve transparency, which has led to various techniques to adjust the middleware to the needs of the applications that make use of it. We discuss some of the more commonly applied techniques, as they affect the organization of the middleware itself.

The actual realization of a distributed system requires that we instantiate and place software components on real machines. There are many different choices that can be made in doing so. The final instantiation of a software architecture is also referred to as a system architecture. In this chapter we will look into traditional centralized architectures in which a single server implements most of the software components, while remote clients can access that server using simple communication means. In addition, we consider decentralized peer-to-peer architectures in which all nodes more or less play equal roles. Many real-world distributed systems are often organized in a hybrid fashion, combining elements from centralized and decentralized architectures. We discuss a few typical examples.

Architectural Styles

We start our discussion on architectures by first considering the logical organization of a distributed system into software components, also referred to as its software architecture. Research on software architectures has matured considerably and it is now commonly accepted that designingor adopting an architecture is crucial for the successful development of large software systems.

For our discussion, the notion of an architectural style is important. Such a style is formulated in terms of components, the way that components are connected to each other, the data exchanged between components, and finally how these elements are jointly configured into a system. A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment. That a component can be replaced, and, in particular, while a system continues to operate, is important. This is due to

the fact that in many cases, it is not an option to shut down a system for maintenance. At best, only parts of it may be put temporarily out of order. Replacing a component can be done only if its interfaces remain untouched.

A somewhat more difficult concept to grasp is that of a connector, which is generally described as a mechanism that mediates communication, coordination, or cooperation among components. For example, a connector can be formed by the facilities for (remote) procedure calls, message passing, or streaming data. In other words, a connector allows for the flow of control and data between components. Using components and connectors, we can come to various configurations, which, in turn, have been classified into architectural styles. Several styles have by now been identified, of which the most important ones for distributed systems are:

- Layered architectures
- Object-based architectures
- Resource-centered architectures
- Event-based architectures

Application Layering

Let us now turn our attention to the logical layering of applications. Considering that a large class of distributed applications is targeted toward supporting user or application access to databases, many people have advocated a distinction between three logical levels, essentially following a layered architectural style:

- The application-interface level
- The processing level
- The data level

In line with this layering, we see that applications can often be constructed from roughly three different pieces: a part that handles interaction with a user or some external application, a part that operates on a database or file system, and a middle part that generally contains the core functionality of the application. This middle part is logically placed at the processing level. In contrast to user interfaces and databases, there are not many aspects common to the processing level. Therefore, we shall give a number of examples to make this level clearer.

As a first example, consider an Internet search engine. Ignoring all the animated banners, images, and other fancy window dressing, the user interface of a search engine can be very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been perfected and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. This information retrieval part is typically placed at the processing level. As a second example, consider a decision support system for stock brokerage. Analogous to a search engine, such a system can be divided into the following three layers:

- A front end implementing the user interface or offering a programming interface to external applications.

- A back end for accessing a database with the financial data.
- The analysis programs between these two.

Analysis of financial data may require sophisticated methods and techniques from statistics and artificial intelligence. In some cases, the core of a financial decision support system may even need to be executed on high- performance computers in order to achieve the throughput and responsiveness that is expected from its users.

As a last example, consider a typical desktop package, consisting of a word processor, a spreadsheet application, communication facilities, and so on. Such "office" suites are generally integrated through a common user interface that supports integrated document management, and operates on files from the user's home directory. In an office environment, this home directory is often placed on a remote file server. In this example, the processing level consists of a relatively large collection of programs, each having rather simple processing capabilities [12].

The data level contains the programs that maintain the actual data on which the applications operate. An important property of this level is that data are often persistent, that is, even if no application is running, data will be stored somewhere for next use. In its simplest form, the data level consists of a file system, but it is also common to use a full-fledged database. Besides merely storing data, the data level is generally also responsible for keeping data consistent across different applications. When databases are being used, maintaining consistency means that metadata such as table descriptions, entry constraints and application-specific metadata are also stored at this level. For example, in the case of a bank, we may want to generate a notification when a customer's credit card debt reaches a certain value. This type of information can be maintained through a database trigger that activates a handler for that trigger at the appropriate moment.

Object based and Service Oriented Architectures

A far more loose organization is followed in object-based architectures. In essence, each object corresponds to what we have defined as a component, and these components are connected through a procedure call mechanism. In the case of distributed systems, a procedure call can also take place over a network, that is, the calling object need not be executed on the same machine as the called object.

Object-based architectures are attractive because they provide a natural way of encapsulating data called an object's state and the operations that can be performed on that data which are referred to as an object's methods into a single entity. The interface offered by an object conceals implementation details, essentially meaning that we, in principle, can consider an object completely independent of its environment. As with components, this also means that if the interface is clearly defined and left otherwise untouched, an object should be replaceable with one having exactly the same interface.

This separation between interfaces and the objects implementing these interfaces allows us to place an interface at one machine, while the object itself resides on another machine. This organization, which is shown in Figure 2.6 is commonly referred to as a distributed object. When a client binds to a distributed object, an implementation of the object's interface, called a proxy, is then loaded into the client's address space. A proxy is analogous to a client stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the

client machine. Incoming invocation requests are first passed to a server stub, which unmarshals them to make method invocations at the object's interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.

The server-side stub is often referred to as a skeleton as it provides the bare means for letting the server middleware access the user-defined objects.In practice, it often contains incomplete code in the form of a language-specific class that needs to be further specialized by the developer. According to the Figure 12A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is not distributed: it resides at a single machine. Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects. In a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object's interfaces. One could argue that object-based architectures form the foundation of encapsulating services into independent units. Encapsulation is the keyword here: the service as a whole is realized as a self-contained entity, although itcan possibly make use of other services. By clearly separating various services such that they can operate independently, we are paving the road toward service-oriented architectures, generally abbreviated as SOAs [13].

In a service-oriented architecture, a distributed application or system is essentially constructed as a composition of many different services. Not all of these services may belong to the same administrative organization. We already came across this phenomenon when discussing cloud computing: it may very well be that an organization running its business application makesuse of storage services offered by a cloud provider. These storage services are logically completely encapsulated into a single unit, of which an interface is made available to customers.

Of course, storage is a rather basic service, but more sophisticated situations easily come to mind. Consider, for example, a Web shop selling goods such as e-books. A simple implementation following the application layering we discussed previously, may consist of an application for processing orders, which, in turn, operates on a local database containing the e-books. Order processing typically involves selecting items, registering and checking the delivery channel, but also making sure that a payment takes place. The latter can be handled by a separate service,run by a different organization, to which a purchasing customer is redirected for the payment, after which the e-book organization is notified so that it can complete the transaction.

In this way, we see that the problem of developing a distributed system is partly one of service composition, and making sure that those services operate in harmony. Indeed, this problem is completely analogous to the enterprise application integration issues. Crucial is, and remains, that each service offers a well-defined (programming) interface. In practice, this also means that each service offers its own interface, in turn, possibly making the composition of services far from trivial.

CHAPTER 9

RESOURCE BASED ARCHITECTURES

Dr. Gopal K Shyam, Professor and HoD,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- gopalkirshna.shyam@presidencyuniversity.in

As an increasing number of services became available over the Web and the development of distributed systems through service composition became more important, researchers started to rethink the architecture of mostly Web-based distributed systems. One of the problems with service composition is that connecting various components can easily turn into an integration nightmare. As an alternative, one can also view a distributed system as a huge collection of resources that are individually managed by components. Resources may be added or removed by applications, and likewise can be retrieved or modified. This approach has now been widely adopted for the Web and is known as Representational State Transfer (REST). There are four key characteristics of what are known as RESTful architecture:

a. Resources are identified through a single naming scheme

b. All services offer the same interface, consisting of at most four operations, as shown in Table 1.

c. Messages sent to or from a service are fully self-described.

Table 1: Illustrated that the Four Operations Available in RESTful Architectures.

Sr. No.	Operation	Description
1.	PUT	Create a new resource
2.	GET	Retrieve the state of a resource in some representation
3.	DELETE	Delete a resource
4.	POST	Modify a resource by transferring a new state

Publish Subscribe Architectures

As systems continue to grow and processes can more easily join or leave, itbecomes important to have an architecture in which dependencies between processes become as loose as possible. A large class of distributed systems have adopted an architecture in which there is a strong separation betweenprocessing and coordination. The idea is to view a system as a collection of autonomously operating processes. In this model, coordination encompasses the communication and cooperation between processes. It provide a taxonomy of coordination models that can be applied equally too many types of distributed systems. Slightly adapting their terminology, we make a distinction between models along two differentdimensions, temporal and referential, as shown in Table 2.

Table 2: Represented that the Examples of Different forms of Coordination.

Sr. No.		Temporally Coupled	Temporally Decoupled
1.	**Referentially Coupled**	Direct	Mailbox
2.	**Referentially Decoupled**	Event based	Shared data space

When processes are temporally and referentially coupled, coordination takes place in a direct way, referred to as direct coordination. The referential coupling generally appears in the form of explicit referencing in communication. For example, a process can communicate only if it knows the name or identifier of the other processes it wants to exchange information with. Temporal coupling means that processes that are communicating will both have to be up and running. In real life, talking over cell phones, is an example of direct communication. A different type of coordination occurs when processes are temporally decoupled, but referentially coupled, which we refer to as mailbox coordination. In this case, there is no need for two communicating processes to be executing at the same time in order to let communication take place. Instead, communication takes place by putting messages in a mail- box. Because it is necessary to explicitly address the mailbox that will hold the messages that are to be exchanged, there is a referential coupling.

The combination of referentially decoupled and temporally coupled systems form the group of models for event-based coordination. In referentially decoupled systems, processes do not know each other explicitly. The only thing a process can do is publish a notification describing the occurrence of an event for example that it wants to coordinate activities, or that it just produced some interesting results. Assuming that notifications come in all sorts and kinds, processes may subscribe to a specific kind of notification. In an ideal event-based coordination model, a published notification will be delivered exactly to those processes that have subscribed to it. However, it is generally required that the subscriber is up-and-running at the time the notification was published [14].

A well-known coordination model is the combination of referentially and temporally decoupled processes, leading to what is known as a shared data space. The key idea is that processes communicate entirely through tuples, which are structured data records consisting of a number of fields, very similar to a row in a database table. Processes can put any type of tuple into the shared data space. In order to retrieve a tuple, a process provides a search pattern that is matched against the tuples. Any tuple that matches is returned. Shared data spaces are thus seen to implement an associative search mechanism for tuples. When a process wants to extract a tuple from the data space, it specifies the values of the fields it is interested in. Any tuple that matches that specification is then removed from the data space and passed to the process.

Shared data spaces are often combined with event-based coordination: a process subscribes to certain tuples by providing a search pattern; when a process inserts a tuple into the data space, matching subscribers are notified. In both cases, we are dealing with publish subscribe architecture, and indeed, the key characteristic feature is that processes have no explicit reference to each other. The difference between a pure event-based architectural style, and that of a shared data space. We have also shown an abstraction of the mechanism by which publishers and subscribers are matched, known as an event bus. An important aspect of publish-subscribe systems is that communication takes place by

describing the events that a subscriber is interested in. As a consequence, naming plays a crucial role. We return to naming later, but for now the important issue is that in many cases, data items are not explicitly identified by senders and receivers.

Let us first assume that events are described by a series of attributes. A notification describing an event is said to be published when it is made available for other processes to read. To that end, a subscription needs to be passed to the middleware, containing a description of the event that the subscriber is interested in. Such a description typically consists of some pairs, which is common for so-called topic-based publish- subscribe systems. As an alternative, in content-based publish-subscribe systems, a subscription may also consist of pairs. In this case, the specified attribute is expected to take on values within a specified range. Descriptions can sometimes be given using all kinds of predicates formulated over the attributes, very similar in nature to SQL-like queries in the case of relational databases. Obviously, the more complex a description is, the more difficult it will be to test whether an event matches a description.

We are now confronted with a situation in which subscriptions need to be matched against notifications. In many cases, an event actually corresponds to data becoming available. In that case, when matching succeeds, there are two possible scenarios. In the first case, the middleware may decide to forward the published notification, along with the associated data, to its current set of subscribers, that is, processes with a matching subscription. As an alternative, the middleware can also forward only a notification at which point subscribers can execute a read operation to retrieve the associated data item.

In those cases in which data associated with an event are immediately forwarded to subscribers, the middleware will generally not offer storage of data. Storage is either explicitly handled by a separate service, or is the responsibility of subscribers. In other words, we have a referentially decoupled, but temporally coupled system.

This situation is different when notifications are sent so that subscribers need to explicitly read the associated data. Necessarily, the middleware will have to store data items. In these situations there are additional operations for data management. It is also possible to attach a lease to a data item such that when the lease expires that the data item is automatically deleted.

Events can easily complicate the processing of subscriptions. To illustrate, consider a subscription such as “notify when room is unoccupied and the door is unlocked.” Typically, a distributed system supporting such subscriptions can be implemented by placing independent sensors for monitoring room occupancy and those for registering the status of a door lock. Following the approach sketched so far, we would need to compose such primitive events into a publishable data item to which processes can then subscribe. Event composition turns out to be a difficult task, notably when the primitive events are generated from sources dispersed across the distributed system.

Clearly, in publish-subscribe systems such as these, the crucial issue is the efficient and scalable implementation of matching subscriptions to notifications. From the outside, publish subscribe architecture provides lots of potential for building very large-scale distributed systems due to the strong decoupling of processes. On the other hand, devising scalable implementations without losing this independence is not a trivial exercise, notably in the case of content-based publish-subscribe systems.

Events can easily complicate the processing of subscriptions to illustrate, consider a subscription such as “notify is unoccupied and the door is unlocked.” Typically, a distributed

system supporting such subscriptions can be implemented by placing independent sensors for monitoring room occupancy and those for registering the status of a door lock. Following the approach sketched so far, we would need to compose such primitive events into a publishable data item to which processes can then subscribe. Event composition turns out to be a difficult task, notably when the primitive events are generated from sources dispersed across the distributed system.

Clearly, in publish-subscribe systems such as these, the crucial issue is theefficient and scalable implementation of matching subscriptions to notifications. From the outside, publish subscribe architecture provides lots of potential for building very large-scale distributed systems due to the strong decoupling of processes. On the other hand, devising scalable implementations without losing this independence is not a trivial exercise, notably in thecase of content-based publish-subscribe systems.

Middleware Organization

In the previous section we discussed a number of architectural styles that are often used as general guidelines to build and organize distributed systems. Let us now zoom into the actual organization of middleware, that is, independent of the overall organization of a distributed system or application. In particular, there are two important types of design patterns that are often applied to the organization of middleware: wrappers and interceptors. Each targets different problems, yet addresses the same goal for middleware: achieving openness. However, it can be argued that the ultimateopenness is achieved when we can compose middleware at runtime.

Interceptors

Conceptually, an interceptor is nothing but a software construct that will break the usual flow of control and allow other application specific codeto be executed. Interceptors are a primary means for adapting middleware to the specific needs of an application. As such, they play an importantrole in making middleware open. To make interceptors generic may requirea substantial implementation effort, as illustrated and it is unclear whether in such cases generality should be preferred over restricted applicability and simplicity. Also, in many cases having only limited interception facilities will improve management of the software andthe distributed system as a whole [15].

To make matters concrete, consider interception as supported in many object-based distributed systems. The basic idea is simple: an object A can call a method that belongs to an object B, while the latter resides on a different machine than A. As we explain in detail later in the book, such a remote-object invocation is carried out in three steps:

a. Object A is offered a local interface that is exactly the same as the interface offered by object B. A calls the method available in that interface.

b. The call by A is transformed into a generic object invocation, made possible through a general object-invocation interface offered by the middleware at the machine where A resides.

c. Finally, the generic object invocation is transformed into a message thatis sent through the transport-level network interface as offered by A's local operating system.

After the first step, the call B.doit (val)is transformed into a generic call such as invoke (B, &doit, val) with a reference to B's method and the parameters that go along with the call.

Now imagine that object B is replicated. In that case, each replica should actually be invoked. This is a clear point where interception can help. What the request-level interceptor will do, is simply call invoke foreach of the replicas. The beauty of this all is that the object A need not be aware of the replication of B, but also the object middleware need not have special components that deal with this replicated call. Only the request-level interceptor, which may be added to the middleware needs to know about B's replication.

In the end, a call to a remote object will have to be sent over the network.In practice, this means that the messaging interface as offered by the local operating system will need to be invoked. At that level, a message-level interceptor may assist in transferring the invocation to the target object. For example, imagine that the parameter val actually corresponds to a huge array of data. In that case, it may be wise to fragment the data into smaller parts to have it assembled again at the destination. Such a fragmentation may improve performance or reliability. Again, the middleware need not be aware of this fragmentation; the lower-level interceptor will transparently handle the rest of the communication with the local operating system.

Modifiable Middleware

What wrappers and interceptors offer are means to extend and adapt the middleware. The need for adaptation comes from the fact that the environment in which distributed applications are executed changes continuously. Changes include those resulting from mobility, a strong variance in the quality-of- service of networks, failing hardware, and battery drainage, amongst others. Rather than making applications responsible for reacting to changes, this task is placed in the middleware. Moreover, as the size of a distributed system increases, changing its parts can rarely be done by temporarily shutting it down. What is needed is being able to make changes on-the-fly.

These strong influences from the environment have brought many de- signers of middleware to consider the construction of adaptive software. In speaking of modifiable middleware to express that middleware may not only need to be adaptive, but that we should be able to purposefully modify it without bringing it down. In this context, interceptors can be thought of offering a means to adapt the standard flow of control. Replacing software components at runtime is an example of modifying a system. And indeed, perhaps one of the most popular approaches toward modifiable middleware is that of dynamically constructing middleware from components.

Component-based design focuses on supporting modifiability through composition. A system may either be configured statically at design time, or dynamically at runtime. The latter requires support for late binding, a technique that has been successfully applied in programming language environments, but also for operating systems where modules can be loaded and unloaded at will. Research is now well underway to automatically select the best implementation of a component during runtime but again, the process remains complex for distributed systems, especially when considering that replacement of one component requires to know exactly what the effect of that replacement on other components will be. In many cases, components are less independent as one may think.

The bottom line is that in order to accommodate dynamic changes to the software that makes up middleware, we need at least basic support to load and unload components at runtime. In addition, for each component explicit specifications of the interfaces it offers, as well the interfaces it requires, are needed. If state is maintained between calls to a component, then further special measures are needed. By-and-large, it should be clear that organizing middleware to be modifiable requires very special attention.

CHAPTER 10

SYSTEM ARCHITECTURE

Dr. Gopal K Shyam, Professor and HoD,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- gopalkirshna.shyam@presidencyuniversity.in

Now that we have briefly discussed some commonly applied architectural styles, let us take a look at how many distributed systems are actually organized by considering where software components are placed. Deciding on software components, their interaction, and their placement leads to an instance of a software architecture, also known as a system architecture. We will discuss centralized and decentralized organizations, as well as various hybrid forms.

Centralized organizations

Despite the lack of consensus on many distributed systems issues, there is one issue that many researchers and practitioners agree upon: thinking in terms of clients that request services from servers helps understanding and managing the complexity of distributed systems. In the following, we first consider a simple layered organization, followed by looking at multi-layered organizations.

Simple Client Server Architecture

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request-reply behavior. Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client. Using a connectionless protocol has the obvious advantage of being efficient.

As long as messages do not get lost or corrupted, the request or reply protocol just sketched works fine. Unfortunately, making the protocol resistant to occasional transmission failures is not trivial. The only thing we can do is possibly let the client resend the request when no reply message comes in. The problem, however, is that the client cannot detect whether the original request message was lost, or that transmission of the reply failed. If the reply was lost, then resending a request may result in performing the operation twice. If theoperation was something like "transfer $10,000 from my bank account," thenclearly, it would have been better that we simply reported an error instead. On the other hand, if the operation was "tell me how much money I have left," it would be perfectly acceptable to resend the request. When an operation can be repeated multiple times without harm, it is said to be idempotent. Since some requests are idempotent and others are not it should be clear thatthere is no single solution for dealing with lost messages.

As an alternative, many client-server systems use a reliable connection- oriented protocol. Although this solution is not entirely appropriate in a local-area network due to relatively low performance, it works perfectly fine in wide-area systems in which communication is inherently unreliable. For example, virtually all Internet application protocols are based on reliable TCP/IP connections. In this case, whenever a client requests a service, it first sets up a connection to the server before sending the request. The server generally uses that same connection to send the reply message, after which the connection is torn down. The trouble may be that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small.

The client-server model has been subject to many debates and controversies over the years. One of the main issues was how to draw a clear distinction between a client and a server. Not surprisingly, there is often no clear distinction. For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables. In such a case, the database server itself only processes the queries.

Multitier Architectures

The distinction into three logical levels as discussed so far, suggests a number of possibilities for physically distributing a client-server application across several machines. The simplest organization is to have only two types of machines:

i. A client machine containing only the programs implementing (part of) the user-interface level.

ii. A server machine containing the rest, that is, the programs implementing the processing and data level.

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with only a convenient graphical interface. There are, however, many other possibilities. As explained many distributed applications are divided into the three layers:

- **i.** User Interface Layer,
- **ii.** Processing Layer,
- **iii.** Data Layer.

One approach for organizing clients and servers is then to distribute these layers across different machines. As a first step, we make a distinction between only two kinds of machines: client machines and server machines, leading to what is also referred to as a (physically) two-tiered architecture. One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Figure 17(a), and give the applications remote control over the presentation of their data. An alternative is to place the entire user-interface software on the client side, as shown in Figure 17(b). In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application residing at the server through an application-specific protocol.

In this model, the front end the client software does no processing other than necessary for presenting the application's interface. Continuing along this line of reasoning, we may also move part of the application to the front end. An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and

where necessary interact with the user. Another example of the organization is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data, but where the advanced support tools such as checking the spelling and grammar execute on the server side.

In many client-server environments, the organizations are particularly popular. These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server. For example, many banking applications run on an end-user's machine where the user prepares transactions and such. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing. Represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages. When distinguishing only client and server machines as we did so far, we miss the point that a server may sometimes need to act as a client, leading to a (physically) three-tiered architecture.

In this architecture, traditionally programs that form part of the processing layer are executed by a separate server, but may additionally be partly distributed across the client and server machines. A typical example of where a three-tiered architecture is used is in transaction processing. A separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers.

Another, but very different example were we often see a three-tiered architecture is in the organization of Web sites. In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place. This application server, in turn, interacts with a database server. For example, an application server may be responsible for running the code to inspect the available inventory of some goods as offered by an electronic bookstore.

Decentralized Organizations: Peer-To-Peer Systems

Multitier client-server architectures are a direct consequence of dividing distributed applications into a user interface, processing components, and data-management components. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitier architecture. We refer to this type of distribution as vertical distribution. The characteristic feature of vertical distribution is that it is achieved by placing logically different components on different machines. The term is related to the concept of vertical fragmentation as used in distributed relational databases, where it means that tables are split column wise, and subsequently distributed across multiple machines.

Again, from a systems-management perspective, having a vertical distribution can help: functions are logically and physically split across multiple machines, where each machine is tailored to a specific group of functions. However, vertical distribution is only one way of organizing client-server applications. In modern architectures, it is often the distribution of the clients and the servers that counts, which we refer to as horizontal distribution. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. In this section we will take a look at a class of modern system architectures that support horizontal distribution, known as peer-to-peer systems.

From a high-level perspective, the processes that constitute a peer-to-peer system are all equal. This means that the functions that need to be carried out are represented by every process that constitutes the distributed system. As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time. Given this symmetric behavior, peer-to-peer architectures evolve around the question how to organize the processes in an overlay network. A network in which the nodes are formed by the processes and the links represent the possible communication channels (which are often realized as TCP connections). A node may not be able to communicate directly with an arbitrary other node, but is required to send messages through the available communication channels. Two types of overlay networks exist: those that are structured and those that are not.

Structured Peer-to-Peer Systems

As its name suggests, in a structured peer-to-peer system the nodes that is processes are organized in an overlay that adheres to a specific, deterministic topology: a ring, a binary tree, a grid, etc. This topology is used to efficiently look up data. Characteristic for structured peer-to-peer systems, is that they are generally based on using a so-called semantic-free index. What this means is that each data item that is to be maintained by the system, is uniquely associated with a key, and that this key is subsequently used as an index. To this end, it is common to use a hash function, so that we get:

key(data item) = hash(data item's value).

The peer-to-peer system as a whole is now responsible for storing (key, value) pairs. To this end, each node is assigned an identifier from the same set of all possible hash values, and each node is made responsible for storing data associated with a specific subset of keys. In essence, the system is thus seen to implement a distributed hash table, generally abbreviated to a DHT.

Following this approach now reduces the essence of structured peer-to- peer systems to being able to look up a data item by means of its key. That is, the system provides an efficient implementation of a function lookup that maps a key to an existing node:

existing node = lookup(key).

This is where the topology of a structured peer-to-peer system plays a crucial role. Any node can be asked to look up a given key, which then boils down to efficiently routing that lookup request to the node responsible for storing the data associated with the given key.

To clarify these matters, let us consider a simple peer-to-peer system with a fixed number of nodes, organized into a hypercube. A hypercube is an n-dimensional cube. To expand the hypercube to five dimensions, we would add another set of two interconnected cubes to the figure, connect the corresponding edges in the two halves, and so on.

Hierarchically Organized peer-to-peer Networks

Notably in unstructured peer-to-peer systems, locating relevant data items can become problematic as the network grows. The reason for this scalability problem is simple: as there is no deterministic way of routing a lookup request to a specific data item, essentially the only technique a node can resort to is searching for the request by means of flooding or randomly walking through the network. As an alternative many peer-to-peer systems have proposed to make use of special nodes that maintain an index of data items.

There are other situations in which abandoning the symmetric nature of peer-to-peer systems is sensible. Consider a collaboration of nodes that offer resources to each other. For example, in a collaborative content delivery network (CDN), nodes may offer storage for hosting copies of Web documents allowing Web clients to access pages nearby, and thus to access them quickly. What is needed is a means to find out where documents can be stored best.In that case, making use of a broker that collects data on resource usage and availability for a number of nodes that are in each other's proximity will allow to quickly select a node with sufficient resources.

Nodes such as those maintaining an index or acting as a broker are generally referred to as super peers. As the name suggests, super peers are often also organized in a peer-to-peer network, leading to a hierarchical organisation. A simple example of such an organization. In this organization, every regular peer, now referred to as a weak peer, is connected as a client to a super peer.All communication from and to a weak peer proceeds through that peer's associated super peer.

In many cases, the association between a weak peer and its super peeris fixed: whenever a weak peer joins the network, it attaches to one of the super peers and remains attached until it leaves the network. Obviously, it is expected that super peers are long-lived processes with high availability. Tocompensate for potential unstable behavior of a super peer, backup schemes can be deployed, such as pairing every super peer with another one and requiring weak peers to attach to both.

Having a fixed association with a super peer may not always be the best solution. For example, in the case of file-sharing networks, it may be betterfor a weak peer to attach to a super peer that maintains an index of files that the weak peer is currently interested in. In that case, chances are bigger that when a weak peer is looking for a specific file, its super peer will know where to find it describe a relatively simple scheme in which the association between weak peer and strong peer can change as weak peers discover better super peers to associate with. In particular, a super peer returning the result of a lookup operation is given preference over other super peers.

As we have seen, peer-to-peer networks offer a flexible means for nodesto join and leave the network. However, with super-peer networks a new problem is introduced, namely how to select the nodes that are eligible tobecome super peer. Super peers that can be used to get started when a weak peer starts from scratch.It appears that the address of each of these default super peers is hard-coded in the Skype software. An address consists of an (IP address, port number)-pair. Each weak peer has a local list of addresses of reachable super peers, calledits host cache. If none of the cached super peers is reachable, it tries to connect to one of the default super peers. The host cache is designed to accommodate a few hundred addresses. To connect to the Skype network, a weak peer is required to establish a TCP connection with a super peer. This is important, notably whena peer is operating behind a firewall, as the super peer can assist inactually contacting that peer.

Let us first consider the situation that one peer A wants to contact another (weak) peer B for which it has a contact address. We need to distinguish three cases, all related to the situation whether or not peers are behind (NATed) firewalls. **Both** A **and** B **are on the public Internet:** Being on the public Internet means that A and B can be directly contacted. In this case, a TCP connection is setup between A and B which is used to exchange control packets. The actualcall takes place using UDP packets between negotiated ports at the caller and callee, respectively.

A operates behind a Firewall, while B is on the Public Internet

In this case, A will first set up a TCP connection to a super peer S, after which S will setup a TCP connection to B. Again, the TCP connections are used to transfer control packets between A and B (via S), after which the actual call will take place through UDP and directly between A and B without flowing through S. However, it seems that S is needed to discover the correct pair of port numbers for the firewall at A to allow for UDP packet exchanges. In principle, this should also be possible with assistance of B.

Both A and B operate Behind a Firewall

This is the most challenging situation, certainly if we also assume that the firewalls restrict UDP traffic. In this case, A will connect to an online super peer S through TCP, after which S will set up a TCP connection to B. These connections are used for exchanging control packets. For the actual call, another super peer is contacted that will act as a **relay** R: A sets up a connection to R, and so will B. All voice traffic is then subsequently forwarded over the two TCP connections, and through R.

How do users find each other? As mentioned, the first thing a weak peer needs to do is establish a TCP connection with a super peer. That super peer is either found in the local host cache, or obtained through one of the default super peers. In order to search for a specific user, a weak peer contacts its super peer, which will return a number of other peers to ask. If that did not lead to any results, the super peer returns another (this time longer) list of peers to which the search request should be forwarded. This process is repeated until the user is found or the requester concludes the user does not exist. Indeed, this can be viewed as a form of policy-based search as we mentioned above. Finding a user means that its address, or that of its associated super peer is returned. In principle, if the searched user is online, a VOIP connection can then be established.

Hybrid Architectures

So far, we have focused on client-server architectures and a number of peer-to- peer architectures. Many distributed systems combine architectural features, as we already came across in super-peer networks. In this section we take a look at some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures.

Edge Server Systems

An important class of distributed systems that is organized according to a hybrid architecture is formed by edge-server systems. These systems are deployed on the Internet where servers are placed "at the edge" of the network. This edge is formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP). Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet. This leads to a general organization like the one.

End users, or clients in general, connect to the Internet by means of an edge server. The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions. More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution. The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates.

This concept of edge-server systems is now often taken a step further: taking cloud computing as implemented in a data center as the core, additional servers at the edge of the network are used to assist in computations and storage, essentially leading to distributed cloud systems. In the case of fog computing, even end-user devices form part of the system and are (partly) controlled by a cloud-service provide.

Collaborative distributed systems

Hybrid structures are notably deployed in collaborative distributed systems. The main issue in many of these systems is to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration. To make matters concrete, let us consider the widely popular **BitTorrent** file-sharing system. BitTorrent is a peer-to-peer file downloading system. Its principal working. The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file. An important design goal was to ensure collaboration. In most file-sharing systems, a significant fraction of participants merely download files but otherwise contribute close to nothing a phenomenon referred to as **free riding**. To prevent this situation, in BitTorrent a file can be downloaded only when the downloading client is providing content to someone else.

To download a file, a user needs to access a global directory, which is generally just one of a few well-known Web sites. Such a directory contains references to what are called torrent files. A torrent file contains the information that is needed to download a specific file. In particular, it contains a link to what is known as a tracker, which is a server that is keeping an accurate account of *active* nodes that have (chunks of) the requested file. An active node is one that is currently downloading the file as well. Obviously, there will be many different trackers, although there will generally be only a single tracker per file (or collection of files).

Once the nodes have been identified from where chunks can be down- loaded, the downloading node effectively becomes active. At that point, it will be forced to help others, for example by providing chunks of the file itis downloading that others do not yet have. This enforcement comes from a very simple rule: if node P notices that node Q is downloading more than it is uploading, P can decide to decrease the rate at which it sends data to Q. This scheme works well provided P has something to download from Q. For this reason, nodes are often supplied with references to many other nodes putting them in a better position to trade data.

Clearly, BitTorrent combines centralized with decentralized solutions. As it turns out, the bottleneck of the system is, not surprisingly, formed by the trackers. In an alternative implementation of BitTorrent, a node also joins a separate structured peer-to-peer system to assist in tracking file downloads. In effect, a central tracker's load is now distributed across the participating nodes, with each node acting as a tracker for a relatively small set of torrent files. The original function of the tracker coordinating the collaborative downloading of a file is retained. However, we note that in many BitTorrent systems used today, the tracking functionality has actually been minimized to a one-time provisioning of peers currently involved in downloading the file. From that moment on, the newly participating peer will communicate only with those peers and no longer with the initial tracker.

CHAPTER 11

THE NETWORK FILE SYSTEM

Dr.S,Senthil Kumar, Professor and HoD,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- senthilkumars@presidencyuniversity.in

Many distributed files systems are organized along the lines of client-server architectures, with Sun Microsystem's **Network File System** (**NFS**) being one of the most widely-deployed ones for Unix systems. Here, we concentrate on NFSv3, the widely-used third version of NFS [Callaghan, 2000] and NFSv4, the most recent, fourth version. We will discuss the differences between them as well.

The basic idea behind NFS is that each file server provides a standardized view of its local file system. In other words, it should not matter how that local file system is implemented; each NFS server supports the same model. This approach has been adopted for other distributed files systems as well. NFS comes with a communication protocol that allows clients to access the files stored on a server, thus allowing a heterogeneous collection of processes, possibly running on different operating systems and machines, to share a common file system.

The model underlying NFS and similar systems is that of a **remote file service**. In this model, clients are offered transparent access to a file system that is managed by a remote server. However, clients are normally unaware of the actual location of files. Instead, they are offered an interface to a file system that is similar to the interface offered by a conventional local file system.In particular, the client is offered only an interface containing various file operations, but the server is responsible for implementing those operations. This model is therefore also referred to as the **remote access model**.

In contrast, in the **upload/download model** a client accesses a file locally after having downloaded it from the server, when the client is finished with the file, it is uploaded back to the server again sothat it can be used by another client. The Internet's FTP service can be usedthis way when a client downloads a complete file, modifies it, and then putsit back. NFS has been implemented for a large number of different operating systems, although the UNIX versions are predominant. For virtually all modern UNIX systems, NFS is generally implemented following the layered architecture.

A client accesses the file system using the system calls provided by its local operating system. However, the local UNIX file system interface is replaced by an interface to the Virtual File System (VFS), which by now is a de facto standard for interfacing to different (distributed) file systems. Virtually all modern operating systems provide VFS, and not doing so more or less forces developers to largely implement huge parts of an operating system when adopting a new file-system structure. With NFS, operations on the VFS interface are either passed to a local file system, or passed to a separate component known as the NFS client, which takes care of handling accessto files stored at a remote server. In NFS, all client-server communication is done through so-called remote procedure calls (RPCs). An RPC is essentiallya standardized way to let a client on machine. A make an ordinary call toa procedure that is implemented on another machine B. The NFS client implements the NFS file system operations as remote procedure calls to the server. Note

that the operations offered by the VFS interface can be different from those offered by the NFS client. The whole idea of the VFS is to hide the differences between various file systems. On the server side, we see a similar organization. The NFS server is responsible for handling incoming client requests. The RPC component at the server converts incoming requests to regular VFS file operations that are subsequently passed to the VFS layer. Again, the VFS is responsible for implementing a local file system in which the actual files are stored.

An important advantage of this scheme is that NFS is largely independent of local file systems as display in Table 3. In principle, it really does not matter whether the operating system at the client or server implements a UNIX file system, a Windows file system, or even an old MS-DOS file system. The only important issue is that these file systems are compliant with the file system model offered by NFS. For example, MS-DOS with its short file names cannot be used to implement an NFS server in a fully transparent way.

Table 3: An incomplete list of NFS file system operations.

Operation	v3	v4	Description
create	Yes	No	Create a regular file
create	No	Yes	Create a nonregular file
link	Yes	Yes	Create a hard link to a file
symlink	Yes	No	Create a symbolic link to a file
mkdir	Yes	No	Create a subdirectory in a given directory
mknod	Yes	No	Create a special file
rename	Yes	Yes	Change the name of a file
remove	Yes	Yes	Remove a file from a file system
rmdir	Yes	No	Remove an empty subdirectory from a directory
open	No	Yes	Open a file
close	No	Yes	Close a file
lookup	Yes	Yes	Look up a file by means of a file name
readdir	Yes	Yes	Read the entries in a directory
readlink	Yes	Yes	Read the path name stored in a symbolic link
getattr	Yes	Yes	Get the attribute values for a file
setattr	Yes	Yes	Set one or more attribute values for a file
read	Yes	Yes	Read the data contained in a file
write	Yes	Yes	Write data to a file

Version 4 allows clients to open and close (regular) files. Opening a nonexistent file has the side effect that a new file is created. To open a file, a client provides a name, along with various values for attributes. For example, a client may specify that a file should be opened for write access. After a file has been successfully opened, a client can access that file by means of its file handle. That handle is also used to close the file, by which the client tells the server that it will no longer need to have access to the file. The server, in turn, can release any state it maintained to provide that client access to the file.

The lookup operation is used to look up a file handle for a given path name.In NFSv3, the lookup operation will not resolve a name beyond what is called a **mount point.** For example, assume that the name /remote/vu refers to a mount point in a naming graph. When resolving the name /remote/vu/mbox, the lookup operation in NFSv3 will return the file handle for the mount point /remote/vu along with the remainder of the path name (i.e., mbox). The client is then required to explicitly mount the filesystem that is needed to complete the name lookup. A file system in this context is the collection of files, attributes, directories, and data blocks that are jointly implemented as a logical block device.

In version 4, matters have been simplified. In this case, lookup will attempt to resolve the entire name, even if this means crossing mount points. Note that this approach is possible only if a file system has already been mounted at mount points. The client is able to detect that a mount point has been crossed by inspecting the file system identifier that is later returned when the lookup completes.

There is a separate operation readdir to read the entries in a directory. This operation returns a list of (name, file handle), pairs along with attribute values that the client requested. The client can also specify how many entries should be returned. The operation returns an offset that can be used in a subsequent call to readdir in order to read the next series of entries. Operation read link is used to read the data associated with a symbolic link. Normally, this data corresponds to a path name that can be subsequently looked up. Note that the lookup operation cannot handle symbolic links. Instead, when a symbolic link is reached, name resolution stops and the client is required to first call read link to find out where name resolution should continue.

Files have various attributes associated with them. Again, there are important differences between NFS version 3 and 4. Typical attributes include the type of the file telling whether we are dealing with a directory, a symbolic link, a special file, etc., the file length, the identifier of the file system that contains the file, and the last time the file was modified. File attributes can be read and set using the operations getattr and setattr, respectively.

Finally, there are operations for reading data from a file, and writing data to a file. Reading data by means of the operation read is completely straightforward.The client specifies the offset and the number of bytes to be read. The client is returned the actual number of bytes that have been read, along with additional status information (e.g., whether the end-of-file has been reached).

Writing data to a file is done using the write operation. The client again specifies the position in the file where writing should start, the number of bytes to be written, and the data. In addition, it can instruct the server to ensure that all data are to be written to stable storage. NFS servers are required to support storage devices that can survive power supply failures, operating system failures, and hardware failures.

The Web

The architecture of Web-based distributed systems is not fundamentally dif- ferent from other distributed systems. However, it is interesting to see how the initial idea of supporting distributed documents has evolved since its inception in 1990s. Documents turned from being purely static and pas- sive to dynamically generated content. Furthermore, in recent years, many organizations have begun supporting services instead of just documents.

Simple Web-based systems

Many Web-based systems are still organized as relatively simple client-server architectures. The core of a Web site is formed by a process that has access to a local file system storing documents. The simplest way to refer to a document is by means of a reference called a Uniform Resource Locator (URL). It specifies where a document is located by embedding the DNS name of its associated server along with a file name by which the server can look up the documentin its local file system. Furthermore, a URL specifies the application-level protocol for transferring the document across the network.

A client interacts with Web servers through a browser, which is responsible for properly displaying a document. Also, a browser accepts input from a user mostly by letting the user select a reference to another document, which it then subsequently fetches and displays. The communication between a browser and Web server is standardized: they both adhere to the HyperTextTransfer Protocol (HTTP). This leads to the overall organization.

Let us zoom in a bit into what a document actually is. Perhaps the simplest form is a standard text file. In that case, the server and browser have barely anything to do: the server copies the file from the local file system and transfers it to the browser. The latter, in turn, merely displays the content of the file ad verbatim without further ado.

More interesting are Web documents that have been marked up, which is usually done in the HyperText Markup Language, or simply **HTML**. In that case, the document includes various instructions expressing how its content should be displayed, similar to what one can expect from any decent word- processing system although those instructions are normally hidden from the end user. For example, instructing text to be emphasized is done by the following markup:

<emph>Emphasize this text</emph>

There are many more of such markup instructions. The point is that the browser understands these instructions and will act accordingly when dis- playing the text.

Documents can contain much more than just markup instructions. In particular, they can have complete programs embedded of which **Javascript** is the one most often deployed. In this case, the browser is warned that there is some code to execute as in:

<script type=''text/javascript''>......................</script>

And as long as the browser as an appropriate embedded interpreter for the specified language, everything between "<script>" and "</script>" will be executed as any other other program. The main benefit of including scripts is that it allows for much better interaction with the end user, including sending information back to the server. (The latter, by the way, has always been supported in HTML through **forms**.) Much more can be said about Web documents, but this is not the place to do so. A good introduction on how to build Web-based applications can be found.

Multitiered architectures

The Web started out as the relatively simple two-tiered client-server system shown in Figure 2.27. By now, this simple architecture has been extended to support much more sophisticated means of documents. In fact, one could justafiably argue that the term “document” is no longer appropriate. For one, most things that we get to see in our browser has been generated on the spot as the result of sending a request to a Web server. Content is stored in a database at the server’s side, along with client-side scripts and such, to be composed on-the-fly into a document which is then subsequently sent to the client’s browser. Documents have thus become completely dynamic.

One of the first enhancements to the basic architecture was support for simple user interaction by means of the Common Gateway Interface or sim- ply **CGI**. CGI defines a standard way by which a Web server can execute a program taking user data as input. Usually, user data come from an HTML form; it specifies the program that is to be executed at the server side, along with parameter values that are filled in by the user. Once the form has been completed, the program’s name and collected parameter values are sent to the serve. When the server sees the request, it starts the program named in the request and passes it the parameter values. At that point, the program simply does its work and generally returns the results in the form of a document that is sent back to the user’s browser to be displayed.

CGI programs can be as sophisticated as a developer wants. After processing the data, the program generates an HTML document and returns that document to the server. The server will then pass the document to the client. An interesting observation is that to the server, it appears as if it is asking the CGI program to fetch a document. In other words, the server does nothing but delegate the fetching of a document to an external program.

The main task of a server used to be handling client requests by simply fetching documents. With CGI programs, fetching a document could be delegated in such a way that the server would remain unaware of whether a document had been generated on the fly, or actually read from the local file system. Note that we have just described a two-tiered organization of server-side software.

However, servers nowadays do much more than just fetching documents. One of the most important enhancements is that servers can also process a document before passing it to the client. In particular, a document may contain a **server-side script**, which is executed by the server when the document has been fetched locally. The result of executing a script is sent along with the rest of the document to the client. The script itself is not sent. In other words, using a server-side script changes a document by essentially replacing the script with the results of its execution. To make matters concrete, take a look at a very simple example of dynamically generating a document. Assume a file is stored at the server with the following content:

```
<strong> <?php echo $_SERVER['REMOTE_ADDR']; ?> </strong>
```

The server will examine the file and subsequently process the PHP code (be- tween “<?php” and “?>”) replacing the code with the address of the requesting client. Much more sophisticated settings are possible, such as accessing a local database and subsequently fetching content from that database to be combined with other dynamically generated content.

CHAPTER 12

AN ELABORATION OF THE PROCESS OF DISTRIBUTED SYSTEM

Mr.Pakruddin, Assistant Professor,
Department of Computer Science and Engineering,
Presidency University, Bangalore, Karnataka, India
Email Id- pakruddin.b@presidencyuniversity.in

In this chapter, we take a closer look at how the different types of processes play a crucial role in distributed systems. The concept of a process originates from the field of operating systems where it is generally defined as a program in execution. From an operating-system perspective, the management and scheduling of processes are perhaps the most important issues to deal with. However, when it comes to distributed systems, other issues turn out to be equally or more important.

We start with extensively discussing threads and their role in distributed systems. As it turns out, threads play a crucial role in obtaining performance in multicore and multiprocessor environments, but also help in structuring clients and servers. There are many cases where we see threads being replaced by processes and using the underlying operating system for guaranteeing protection and facilitating communication. Nevertheless, when performance is at stake, threads continue to play an important role [16].

In recent years, the concept of virtualization has regained much popu- larity. Virtualization allows an application, and possibly also its complete environment including the operating system, to run concurrently with other applications, but highly independent of the underlying hardware and plat- forms, leading to a high degree of portability. Moreover, virtualization helps in isolating failures caused by errors or security problems. It is an important concept for distributed systems, and we pay attention to it in a separate section.

Client-server organizations are important in distributed systems. In this chapter, we take a closer look at typical organizations of both clients and servers. We also pay attention to general design issues for servers, including those typically used in object-based distributed systems. A widely used Web server is Apache, to which we pay separate attention. The organization of server clusters remains important, especially when they need to collaborate provide the illusion of a single system. We will discuss examples of how to achieve this perspective, including wide-area servers like PlanetLab.

An important issue, especially in wide-area distributed systems, is moving processes between different machines. Process migration or more specifically, code migration, can help in achieving scalability, but can also help to dynamically configure clients and servers. What is actually meant by code migration and what its implications are is also discussed in this chapter.

Threads

Although processes form a building block in distributed systems, practice indicates that the granularity of processes as provided by the operating systems on which distributed systems are built is not sufficient. Instead, it turns out that having a finer granularity in the form of multiple threads of control per process makes it much easier to build distributed

applications and to get better performance. In this section, we take a closer look at the role of threads in distributed systems and explain why they are so important. More on threads and how they can be used to build applications can be found in [Lewis and Berg, 1998; Stevens, 1999; Robbins and Robbins, 2003]. Herlihy and Shavit [2008] is highly recommended to learn more about multithreaded concurrent programming in general.

Introduction to threads

To understand the role of threads in distributed systems, it is important to un- derstand what a process is, and how processes and threads relate. To execute a program, an operating system creates a number of virtual processors, each one for running a different program. To keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc. Jointly, these entries form a process context.

A process context can be viewed as the software analog of the hardware's processor context. The latter consists of the minimal information that is automatically stored by the hardware to handle an interrupt, and to later return to where the CPU left off. The processor context contains at least the program counter, but sometimes also other register values such as the stack pointer.

A process is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors. An important issue is that the operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior. In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent. Usually, the operating system requires hardware support to enforce this separation.

This concurrency transparency comes at a price. For example, each time a process is created, the operating system must create a complete independent address space. Allocation can mean initializing memory segments by, for example, zeroing a data segment, copying the associated program into a text segment, and setting up a stack for temporary data. Likewise, switching the CPU between two processes may require some effort as well. Apart from saving the data as currently stored in various registers including the program counter and stack pointer, the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation lookaside buffer (TLB). In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.

Like a process, a thread executes its own piece of code, independently from other threads. However, in contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation. Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads. In particular, a thread context often consists of nothing more than the processor context, along with some other information for thread management. For example, a thread system may keep track of the fact that a thread is currently blocked on a mutex variable, so as not to select it for execution. Information that is not strictly necessary to manage multiple threads is generally ignored. For this reason, protecting data against inappropriate access by threads within a single process is left entirely to application developers. We thus see that a processor context is contained in a thread context, and that, in turn, a thread context is contained in a process context.

There are two important implications of deploying threads as we just sketched. First of all, the performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, in many cases, multithreading even leads to a performance gain. Second, because threads are not automatically protected against each other the way processes are, development of multithreaded applications requires additional intellectual effort. Proper design and keeping things simple, as usual, help a lot. Unfortunately, current practice does not demonstrate that this principle is equally well understood.

Thread usage in no distributed systems

Before discussing the role of threads in distributed systems, let us first consider their usage in traditional, no distributed systems. There are several benefits to multithreaded processes that have increased the popularity of using thread systems. The most important benefit comes from the fact that in a single-threaded process, whenever a blocking system call is executed, the process as a whole is blocked. To illustrate, consider an application such as a spreadsheet program, and assume that a user continuously and interactively wants to change values. An important property of a spreadsheet program is that it maintains the functional dependencies between different cells, often from different spreadsheets.

Therefore, whenever a cell is modified, all dependent cells are automatically updated. When a user changes the value in a single cell, such a modification can trigger a large series of computations. If there is only a single thread of control, computation cannot proceed while the program is waiting for input. Likewise, it is not easy to provide input while dependencies are being calculated. The easy solution is to have at least two threads of control: one for handling interaction with the user and one for updating the spreadsheet. In the meantime, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.

Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor or multicore system. In that case, each thread is assigned to a different CPU or core while shared data are stored in shared main memory. When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, albeit slower. Multithreading for parallelism is becoming increasingly important with the availability of relatively cheap multiprocessor and multicore computers. Such computer systems are typically used for running servers in client-server applications, but are by now also extensively used in devices such as smartphones[17].

Multithreading is also useful in the context of large applications. Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process. This approach is typical for a UNIX environment. Cooperation between programs is implemented by means of interposes communication (IPC) mechanisms. For UNIX systems, these mechanisms typically include (named) pipes, message queues, and shared memory segments. The major drawback of all IPC mechanisms is that communication often requires relatively extensive context switching, shown at three different points.

Instead of using processes, an application can also be constructed such that different parts are executed by separate threads. Communication between those parts is entirely dealt with by using shared data. Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

Finally, there is also a pure software engineering reason to use threads: many applications are simply easier to structure as a collection of cooperating threads. Think of applications that need to perform several more or less independent tasks, like our spreadsheet example discussed previously.

Thread Implementation

Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables. There are basically two approaches to implement a thread package. The first approach is to construct a thread library that is executed entirely in user space. The second approach is to have the kernel be aware of threads and schedule them.

A user-level thread library has a number of advantages. First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used. Both operations are cheap.

A second advantage of user-level threads is that switching thread context can often be done in just a few instructions. Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, do CPU accounting, and so on. Switching thread context is done when two threads need to synchronize, for example, when entering a section of shared data. However, as discussed in Note 3.1, much of the overhead of context switching is caused by perturbation memory caches.

A major drawback of user-level threads comes from deploying the many- to-one threading model: multiple threads are mapped to a single schedulable entity. As a consequence, the invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process. As we explained, threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime. For such applications, user-level threads are of no help.

These problems can be mostly circumvented by implementing threads in the operating system's kernel, leading to what is known as the one-to-one threading model in which every thread is a schedulable entity. The price to pay is that every thread operation, will have to be carried out by the kernel, requiring a system call. Switching thread contexts may now become as expensive as switching process contexts. However, in light of the fact that the performance of context switching is generally dictated by ineffective use of memory caches, and not by the distinction between the many-to-one or one-to-one threading model, many operating systems now offer the latter model, if only for its simplicity.

Meanwhile, other LWPs may be looking for other runnable threads as well. If a thread needs to block on a mutex or condition variable, it does the necessary ad- ministration and eventually calls the scheduling routine. When another runnable thread has been found, a context switch is made to that thread. The beauty of all this is that the LWP executing the thread need not be informed: the context switch is implemented completely in user space and appears to the LWP as normal program code.

Now let us see what happens when a thread does a blocking system call. In that case, execution changes from user mode to kernel mode, but still continues in the context of the

current LWP. At the point where the current LWP can no longer continue, the operating system may decide to switch context to another LWP, which also implies that a context switch is made back to user mode. The selected LWP will simply continue where it had previously left off.

There are several advantages to using LWPs in combination with a user- level thread package. First, creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all. Second, provided that a process has enough LWPs, a blocking system call will not suspend the entire process. Third, there is no need for an application to know about the LWPs. All it sees are user-level threads. Fourth, LWPs can be easily used in multiprocessing environments by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application. The only drawback of lightweight processes in combination with user-level threads is thatwe still need to create and destroy LWPs, which is just as expensive as with kernel-level threads. However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system[18].

As a final note, it is important to realize that using threads is one wayof organizing simultaneous and concurrent executions within an application. In practice, we often see that applications are constructed as a collection of concurrent *processes*, jointly making use of the inter-poses facilities offered by an operating system. A good example of this approach is the organization of the Apache Web server,which, by default, starts with a handful of processes for handling incoming requests. Each process forms a single-threaded instantiation of the server, yet is capable of communicating with other instances through fairly standard means.

Threads in Distributed Systems

An important property of threads is that they can provide a convenient meansof allowing blocking system calls without blocking the entire process in which the thread is running. This property makes threads particularly attractive touse in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time. We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.

CHAPTER 13

MULTITHREADED CLIENTS

Ramesh Chandra Tripathi, Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- rctripathig@gmail.com

To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to conceal long intercrosses message propagation times. The round-trip delay in a wide-area network can easily be in the order of hundreds of milliseconds, or sometimes even seconds. The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in Web browsers. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component. Setting up a connection as well as reading incoming data are inherently blocking operations. When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.

A Web browser often starts with fetching the HTML page and subsequently displays it. To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in. While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images. The latter are displayed as they are brought in. The user need thus not wait until all the components of the entire page are fetched before the page is made available.

In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably. As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming thata blocking call does not suspend the entire process. Meanwhile, the user notices only delays in the display of images and such, but can otherwise browse through the document.

There is another important benefit to using multithreaded Web browsersin which several connections can be opened simultaneously. In the previousexample, several connections were set up to the same server. If that server isheavily loaded, or just plain slow, no real performance improvements will benoticed compared to pulling in the files that make up the page strictly one after the other.

However, in many cases, Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents. The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique. When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively

establishing that the entire Web document is fully displayed in a much shorter time than with a no replicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

Multithreaded Servers

Although there are important benefits to multithreaded clients, the main use of multithreading in distributed systems is found at the server side. Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uniprocessor systems. However, with modern multicore processors, multithreading for parallelism is an obvious path to follow.

To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk. The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. One possible, and particularly popular organization. Here one thread, the **dispatcher**, reads incoming requests for a file operation. The requests are sent by clients to a well-known end point for this server. After examining the request, the server chooses an idle **worker thread** and hands it the request.

The worker proceeds by performing a blocking read on the *local* file system, which may cause the thread to be suspended until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

Now consider how the file server might have been written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled. In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests per time unit can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way [19].

So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. A third alternative is to run the server as a big single-threaded finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the in-memory cache, fine, but if not, the thread must access the disk. However, instead of issuing a blocking disk operation, the thread schedules an asynchronous disk operation for which it will be later interrupted by the operating system. To make this work, the thread will record the status of the request namely, that it has a pending disk operation, and continues to see if there were any other incoming requests that require its attention. Once a pending disk operation has been completed, the operating system will notify the thread, who will then, in due time, look up the status of the associated request and continue processing it.

Eventually, a response will be sent to the originating client, again using a no blocking call to send a message over the network. In this design, the “sequential process” model that we had in the first two cases is lost. Every time the thread needs to do a blocking operation, it needs to record exactly where it was in processing the request, possibly also storing additional state.

Once that has been done, it can start the operationand continue with other work. Other work means processing newly arrived requests, or post processing requests for which a previously started operation has completed. Of course, if there is no work to be done, the thread may indeed block. In effect, we are simulating the behavior of multiple threads and their respective stacks the hard way. The process is being operated asa finite-state machine that gets an event and then reacts to it, depending on what is in it.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls and still achieve parallelism. Blocking system calls make programming easier as theyappear as just normal procedure calls. In addition, multiple threads allow forparallelism and thus performance improvement. The single-threaded serverretains the ease and simplicity of blocking system calls, but may severely hinder performance in terms of number of requests that can be handledper time unit. The finite-state machine approach achieves high performance through parallelism, but uses no-blocking calls, which is generally hard to program and thus to maintain. These models are summarized in Table 1.

Table 1: Represented that the three ways to construct a Server.

Sr. No.	Model	Characteristics
1.	Multithreading	Parallelism, blocking system calls
2.	Single-threaded process	No parallelism, blocking system calls
3.	Finite-state machine	Parallelism, no blocking system calls

Again, note that instead of using threads, we can also use multiple processes to organize a server leading to the situation that we actually have a multiprocessor server. The advantage is that the operating system can offer more protection against accidental access to shared data. However, if processes need to communicate a lot, we may see a noticeable adverse effect onperformance in comparison to using threads.

Virtualization

Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us to build pieces of programs that appear to be executed simultaneously. On a single-processor computer, this simultaneousexecution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created. This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to whatis known as resource virtualization. This virtualization has been applied for many decades, but has received renewed interest as computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware.

Principle of Virtualization

In practice, every distributed computer system offers a programming interface to higher-level software. There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems. In its essence, virtualization deals with

extending or replacing an existing interface so as to mimic the behavior of another system. We will come to discuss technical details on virtualization shortly, but let us first concentrate on why virtualization is important.

Virtualization and Distributed Systems

One of the most important reasons for introducing virtualization back in the1970s, was to allow legacy software to run on expensive mainframe hardware. The software not only included various applications, but in fact also the operating systems they were developed for. This approach toward supporting legacy software has been successfully applied on the IBM 370 mainframes and their successors that offered a virtual machine to which different operating systems had been ported.

As hardware became cheaper, computers became more powerful, and the number of different operating system flavors was reducing, virtualization became less of an issue. However, matters have changed again since the late 1990s. First, while hardware and low-level systems software change reasonably fast, software at higher levels of abstraction e.g., middleware andapplications, are often much more stable. In other words, we are facing the situation that legacy software cannot be maintained in the same pace as the platforms it relies on. Virtualization can help here by porting the legacy interfaces to the new platforms and thus immediately opening up the latter for large classes of existing programs.

Equally important is the fact that networking has become completely pervasive. It is hard to imagine that a modern computer is not connected to a network. In practice, this connectivity requires that system administrators maintain a large and heterogeneous collection of server computers, each one running very different applications, which can be accessed by clients.At the same time the various resources should be easily accessible to these applications. Virtualization can help a lot: the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries *and* operating system, which, in turn, run on a common platform.

This last type of virtualization provides a high degree of portability and flexibility. For example, in order to realize content delivery networks that can easily support replication of dynamic content, have argued that management becomes much easier if edge servers would support virtualization, allowing a complete site, including its environment to be dynamically copied. These arguments are still valid, and indeed, portability is perhaps the most important reason why virtualization plays such a key role in many distributed systems[20].

Types of virtualization

There are many different ways in which virtualization can be realized. An overview of these various approaches is described by Smith and Nair [2005a]. To understand the differences in virtualization, it is important to realize thatcomputer systems generally offer four different types of interfaces, at three different levels:

i. An interface between the hardware and software, referred to as the **instruction set architecture** (**ISA**), forming the set of machine instructions. This set is divided into two subsets:

1. Privileged instructions, which are allowed to be executed only by the operating system.
2. General instructions, which can be executed by any program.
   - **i.** An interface consisting of **system calls** as offered by an operating system.
   - **ii.** An interface consisting of library calls, generally forming what is known as an **application programming interface** (**API**). In many cases, the aforementioned system calls are hidden by an API.

The essence of virtualization is to mimic the behavior of these interfaces. Virtualization can take place in two different ways. First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications. Instructions can be interpreted as is the case for the Java runtime environment, but could also be emulated as is done for running Windows applications on UNIX platforms. Note that in the latter case, the emulator will also have to mimic the behavior of system calls, which has proven to be generally far from trivial. This type of virtualization, leads to what call a process virtual machine, stressing that virtualization is only for a single process.

An alternative approach toward virtualization, is to provide a system that is implemented as a layer shielding the original hardware, but offering the complete instruction set of that same as an interface. This leads to what is known as a native virtual machine monitor. It is called native because it is implemented directly on top of the underlying hardware. Note that the interface offered by a virtual machine monitor can be offered simultaneously to different programs. As a result, it is now possible to have multiple, and different guest operating systems run independently and concurrently on the same platform.

A native virtual machine monitor will have to provide and regulate access to various resources, like external storage and networks. Like any operating system, this implies that it will have to implement device drivers for those resources. Rather than doing all this effort anew, a hosted virtual machine monitor will run on top of a trusted host operating system. In this case, the virtual machine monitor can make use of existing facilities provided by that host operating system. It will generally have to be given special privileges instead of running as a user-level application. Using a hosted virtual machine monitor is highly popular in modern distributed systems such as data centers and clouds.

As virtual machines are becoming increasingly important in the context of reliability and security for (distributed) systems. As they allow for the isolation of a complete application and its environment, a failure caused by an error or security attack need no longer affect a complete machine. In addition, as we also mentioned before, portability is greatly improved as virtual machines provide a further decoupling between hardware and software, allowing a complete environment to be moved from one machine to another.

Application of Virtual Machines to Distributed Systems

From the perspective of distributed systems, the most important application of virtualization lies in cloud computing. As we already mentioned cloud providers offer roughly three different types of services:

a. **Infrastructure-as-a-Service** (**IaaS**) covering the basic infrastructure

b. **Platform-as-a-Service** (**PaaS**) covering system-level services

c. **Software-as-a-Service** (**SaaS**) containing actual applications

Virtualization plays a key role in IaaS. Instead of renting out a physical machine, a cloud provider will rent out a virtual machine (monitor) that may, or may not, be sharing a physical machine with other customers. The beauty of virtualization is that it allows for almost complete isolation between customers, who will indeed have the illusion that they have just rented a dedicated physical machine. Isolation is, however, never complete, if only for the fact that the actual physical resources are shared, in turn leading to observable lower performance.

To make matters concrete, let us consider the Amazon Elastic ComputeCloud, or simply EC2. EC2 allows one to create an environment consisting of several networked virtual servers, thus jointly forming the basis of a distributed system. To make life easy, there is a (large) number of pre- configured machine images available, referred to as Amazon Machine Images, or simply AMIs. An AMI is an installable software package consisting of an operating-system kernel along with a number of services. An example of a simple, basic AMI is a LAMP image, consisting of a Linux kernel, the Apache Web server, a MySQL database system, and PHP libraries.

More elaborate images containing additional software are also available, as well as images based on other UNIX kernels or Windows. In this sense, an AMI is essentially the same as a boot disk although there are few important differences to which we return shortly. An EC2 customer needs to select an AMI, possibly after adapting orconfiguring one. An AMI can then be launched resulting in what is called an EC2 instance: the actual virtual machine that can be used to host a customer's applications. An important issue is that a customer will hardly ever know exactly where an instance is actually being executed. Obviously, it is runningon a single physical machine, but where that machine is located remains hidden.

To communicate, each instance obtains two IP addresses: a private one that can be used for internal communication between different instances, making use of EC2's internal networking facilities, and a public IP address allowing any Internet clients to contact an instance. The public address is mapped to the private one using standard Network Address Translation (NAT) technology. A simple way to manage an instance is to make use of an SSH connection, for which Amazon provides the means for generating the appropriate keys. The EC2 environment in which an instance is executed provides differentlevels of the following services:

a. **CPU**: allows to select the number and type of cores, including GPUs

b. **Memory**: defines how much main memory is allocated to an instance

c. **Storage**: defines how much local storage is allocated

d. **Platform**: distinguishes between 32-bit or 64-bit architectures

e. **Networking**: sets the bandwidth capacity that can be used

In addition, extra resources can be requested such as an additional networking interface. The local storage that comes with an instance is *transient*: when the instance stops, all the data stored locally is lost. In order to prevent data loss,a customer will need to explicitly save data to persistent store, for example, by making use of Amazon's Simple Storage Service (S3). An alternative is to attach a storage device that is mapped to Amazon's Elastic Block Store (Amazon EBS). Again, this is yet another service, but one that can be used in the form of a virtual block device that is simply mounted as one would mount an additional hard disk. When an instance is stopped, all data that was stored on EBS will persist. And just as one would expect, an EBS device can be (re)mounted to any other instance as well.

It should be clear by now that, without having gone into any significant level of detail, the IaaS as offered by EC2 allows a customer to create a (potentially large) number of virtual machines, each configured with resources as needed, and capable of exchanging messages through an IP network. In addition, these virtual machines can be accessed from anywhere over the Internet provided a client has the proper credentials. As such, Amazon EC2, like many other IaaS providers, offers the means to configure a complete distributed system consisting of networked virtual servers and running customer-supplied distributed applications. At the same time, those customers will not need to maintain any physical machine, which by itself is often already a huge gain as we will encounter at several occasions throughout this text. One can indeed argue that virtualization lies at the core of modern cloud computing.

Clients

In the previous chapters we discussed the client-server model, the roles of clients and servers, and the ways they interact. Let us now take a closer lookat the anatomy of clients and servers, respectively. We start in this section with a discussion of clients. Servers are discussed in the next section.

Networked User Interfaces

A major task of client machines is to provide the means for users to interactwith remote servers. There are roughly two ways in which this interactioncan be supported. First, for each remote service the client machine will have a separate counterpart that can contact the service over the network. A typical example is a calendar running on a user's smartphone that needs to synchronize with a remote, possibly shared calendar. In this case, an application-level protocol will handle the synchronization.

A second solution is to provide direct access to remote services by offering only a convenient user interface. Effectively, this means that the client machine is used only as a terminal with no need for local storage, leadingto an application-neutral solution. In the case of networked user interfaces, everything is processed and stored at the server.This thin-client approach has received much attention with the increase of Internet connectivity and the use of mobile devices. Thin-client solutions arealso popular as they ease the task of system management as display.

CHAPTER 14

CLIENT-SIDE SOFTWARE FOR DISTRIBUTION TRANSPARENCY

Tushar Mehrotra, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- tushar.cs17@nitp.ac.in

Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc. In these cases, the user interface is a relatively small part of the client software, in contrast to the local processing and communication facilities. Besides the user interface and other application-related software, client soft- ware comprises components for achieving distribution transparency. Ideally, a client should not be aware that it is communicating with remote processes. In contrast, distribution is often less transparent to servers for reasons of performance and correctness.

Access transparency is generally handled through the generation of a **client stub** from an interface definition of what the server has to offer. The stub provides the same interface as the one available at the server, but hides the possible differences in machine architectures, as well as the actual communication. The client stub transforms local calls to messages that are sent to the server, and *vice versa* transforms messages from the server to return values as one would expect when calling an ordinary procedure.

There are different ways to handle location, migration, and relocation transparency. Using a convenient naming system is crucial. In many cases, cooperation with client-side software is also important. For example, when a client is already bound to a server, the client can be directly informed when the server changes location. In this case, the client's middleware can hide the server's current network location from the user, and also transparently rebind to the server if necessary. At worst, the client's application may notice a temporary loss of performance.

In a similar way, many distributed systems implement replication transparency by means of client-side solutions. For example, imagine a distributed system with replicated servers, such replication can be achieved by forwarding a request to each replica. Client-side software can transparently collect all responses and pass a single response to the client application.

Regarding failure transparency, masking communication failures with a server is typically done through client middleware. For example, client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. There are even situations in which the client middleware returns data it had cached during a previous session, as is sometimes done by Web browsers that fail to connect to a server. Finally, concurrency transparency can be handled through special intermediate servers, notably transaction monitors, and requires less support from client software.

Servers

Let us now take a closer look at the organization of servers. In the following pages, we first concentrate on a number of general design issues for servers, followed by a discussion on server clusters.

General Design Issues

A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

Concurrent versus Iterative Servers

There are several ways to organize servers. In the case of an iterative server, the server itself handles the request and, if necessary, returns a response to the requesting client. A concurrent server does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request. This approach is followed in many Unix systems. The thread or process that handles the request is responsible for returning a response to the requesting client.

Contacting a server: end points

Another issue is where clients contact a server. In all cases, clients send requests to an **end point**, also called a **port**, at the machine where the server is running. Each server listens to a specific end point. How do clients know the end point of a service? One approach is to globally assign end points for well-known services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80. These end points have been assigned by the Internet Assigned Numbers Authority (IANA), and are documented in [Reynolds and Postel, 1994]. With assigned end points, the client needs to find only the network address of the machine where the server is running. Name services can be used for that purpose.

There are many services that do not require a preassigned end point. For example, a time-of-day server may use an end point that is dynamically assigned to it by its local operating system. In that case, a client will first have to look up the end point. One solution is to have a special daemon running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. The daemon itself listens to a well-known end point. A client will first contact the daemon, request the end point, and then contact the specific server. It is common to associate an end point with a specific service. However, actually implementing each service by means of a separate server may be a waste of resources. For example, in a typical UNIX system, it is common to have lots of servers running simultaneously, with most of them passively waiting until a client request comes in. Instead of having to keep track of so many passive processes, it is often more efficient to have a single super server listening to each end point associated with a specific service. For example, the inetd daemon in UNIX listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to handle it. That process will exit when finished.

Interrupting a Server

Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted. For example, consider a user who has just decided to upload a huge file to an FTP server. Then, suddenly realizing that it is the wrong file, he wants to interrupt the server to cancel further data transmission. There are several ways to do this. One approach that works only too well in the current Internet and is sometimes the only alternative is for the user to abruptly exit the client application which will automatically break the connection to the server, immediately restart it, and pretend nothing happened. The server will eventually tear down the old connection, thinking the client has probably crashed.

A much better approach for handling communication interrupts is to develop the client and server such that it is possible to send out-of-band data, which is data that is to be processed by the server before any other data from that client. One solution is to let the server listen to a separate control end point to which the client sends out-of-band data, while at the same time listening with a lower priority to the end point through which the normal data passes.

Another solution is to send out-of-band data across the same connection through which the client is sending the original request. In TCP, for example, it is possible to transmit urgent data. When urgent data are received at the server, the latter is interrupted, after which it can inspect the data and handle them accordingly.

Stateless Versus Stateful Servers

A final, important design issue, is whether or not the server is stateless. A stateless server does not keep information on the state of its clients, and can change its own state without having to inform any client. A Web server, for example, is stateless. It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or most often for fetching a file. When the request has been processed, the Web server forgets the client completely. Likewise, the collection of files that a Web server manages possibly in cooperation with a file server, can be changed without clients having to be informed.

Note that in many stateless designs, the server actually does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server. For example, a Web server generally logs all client requests. This information is useful, for example, to decide whether certain documents should be replicated, and where they should be replicated to. Clearly, there is no penalty other than perhaps in the form of suboptimal performance if the log is lost.

A particular form of a stateless design is where the server maintains what is known as soft state. In this case, the server promises to maintain state on behalf of the client, but only for a limited time. After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client. An example of this type of state is a server promising to keep a client informed about updates, but only for a limited time. After that, the client is required to poll the server for updates. Soft-state approaches originate from protocol design in computer networks, but can be equally applied to server design.

In contrast, a stateful server generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update

operations. Such a server would maintain a table containing (client, file) entries. Such a table allows theserver to keep track of which client currently has the update permissions onwhich file, and thus possibly also the most recent version of that file.

This approach can improve the performance of read and write operations as perceived by the client. Performance improvement over stateless serversis often an important benefit of stateful designs. However, the example also illustrates the major drawback of stateful servers. If the server crashes, it hasto recover its table of (client, file) entries, or otherwise it cannot guarantee that it has processed the most recent updates on a file. In general, a stateful server needs to recover its entire state as it was just before the crash. Enabling recovery can introduce considerable complexity, as we discuss in. Chapter 8In a stateless design, no special measures need to be taken at all for a crashed server to recover. It simply starts running again, and waits for client requeststo come in.

The example above is typical for session state: it is associated with a series of operations by a single user and should be maintained for a sometime, but not indefinitely. As it turns out, session state is often maintained in three-tiered client-server architectures, where the application server actually needs to access a database server through a series of queries before being able to respond to the requesting client. The issue here is that no real harm is done if session state is lost, provided that the client can simply re-issue the original request. This observation allows for simpler and less reliable storage of state. What remains for permanent state is typically information maintained in databases, such as customer information, keys associated with purchased software, etc. However, for most distributed systems, maintaining session state already implies a stateful design requiring special measures when failures do happen and making explicit assumptions about the durability of state storedat the server. We will return to these matters extensively when discussingfault tolerance.

When designing a server, the choice for a stateless or stateful design should not affect the services provided by the server. For example, if files have to be opened before they can be read from, or written to, then a stateless server should one way or the other mimic this behavior. A common solution is thatthe server responds to a read or write request by first opening the referred file, then does the actual read or write operation, and immediately closes thefile again.

In other cases, a server may want to keep a record on a client's behavior so that it can more effectively respond to its requests. For example, Web servers sometimes offer the possibility to immediately direct a client to his favorite pages. This approach is possible only if the server has history information on that client. When the server cannot maintain state, a common solution is then to let the client send along additional information on its previous accesses. In the case of the Web, this information is often transparently stored by the client's browser in what is called a cookie, which is a small piece of data containing client-specific information that is of interest to the server. Cookies are never executed by a browser; they are merely stored.

The first time a client accesses a server, the latter sends a cookie along with the requested Web pages back to the browser, after which the browser safely tucks the cookie away. Each subsequent time the client accesses the server, its cookie for that server is sent along with the request.

Object Servers

Let us take a look at the general organization of object servers needed fordistributed objects. The important difference between a general object serverand other (more traditional) servers

is that an object server by itself does not provide a specific service. Specific services are implemented by the objects that reside in the server. Essentially, the server provides only the means to invoke local objects, based on requests from remote clients. As a consequence, it is relatively easy to change services by simply adding and removing objects.

An object server thus acts as a place where objects live. An object consists of two parts: data representing its state and the code for executing its methods. Whether or not these parts are separated, or whether method implementations are shared by multiple objects, depends on the object server. Also, there are differences in the way an object server invokes its objects. For example, in a multithreaded server, each object may be assigned a separate thread, or a separate thread may be used for each invocation request. These and other issues are discussed next.

For an object to be invoked, the object server needs to know which code to execute, on which data it should operate, whether it should start a separate thread to take care of the invocation, and so on. A simple approach is to assume that all objects look alike and that there is only one way to invokean object. Unfortunately, such an approach is generally inflexible and often unnecessarily constrains developers of distributed objects.

A much better approach is for a server to support different policies. Con- sider, for example, a transient object: an object that exists only as long as its server exists, but possibly for a shorter period of time. An in-memory, read-only copy of a file could typically be implemented as a transient object. Likewise, a calculator could also be implemented as a transient object. A reasonable policy is to create a transient object at the first invocation requestand to destroy it as soon as no clients are bound to it anymore.

The advantage of this approach is that a transient object will need a server'sresources only as long as the object is really needed. The drawback is that an invocation may take some time to complete, because the object needs to be created first. Therefore, an alternative policy is sometimes to create all transient objects at the time the server is initialized, at the cost of consumingresources even when no client is making use of the object.

In a similar fashion, a server could follow the policy that each of its objectsis placed in a memory segment of its own. In other words, objects share neither code nor data. Such a policy may be necessary when an object implementation does not separate code and data, or when objects need to be separated for security reasons. In the latter case, the server will need to provide specialmeasures, or require support from the underlying operating system, to ensure that segment boundaries are not violated.

The alternative approach is to let objects at least share their code. For example, a database containing objects that belong to the same class can be efficiently implemented by loading the class implementation only once into the server. When a request for an object invocation comes in, the server needonly fetch that object's state and execute the requested method. Likewise, there are many different policies with respect to threading. Thesimplest approach is to implement the server with only a single thread of control. Alternatively, the server may have several threads, one for each of its objects. Whenever an invocation request comes in for an object, the server passes the request to the thread responsible for that object. If the thread is currently busy, the request is temporarily queued.

The advantage of this approach is that objects are automatically protected against concurrent access: all invocations are serialized through the single thread associated with the object. Neat and simple. Of course, it is also possible to use a separate thread for each invocation request, requiring that objects should have already been protected against concurrent access.

Independentof using a thread per object or thread per method is the choice of whether threads are created on demand or the server maintains a pool of threads. Generally there is no single best policy.

Decisions on how to invoke an object are commonly referred to as activetion policies, to emphasize that in many cases the object itself must first be brought into the server's address space (i.e., activated) before it can actuallybe invoked. What is needed then is a mechanism to group objects per policy. Such a mechanism is sometimes called an object adapter, or alternatively an object wrapper. An object adapter can best be thought of as software implementing a specific activation policy. The main issue, however, is that object adapters come as generic components to assist developers of distributed objects, and which need only to be configured for a specific policy.

An object adapter has one or more objects under its control. Because a server should be capable of simultaneously supporting objects that require different activation policies, several object adapters may reside in the same server. When an invocation request is delivered to the server, the request is first dispatched to the appropriate object adapter. An important observation is that object adapters are unaware of the specific interfaces of the objects they control. Otherwise, they could never be generic.

The only issue that is important to an object adapter is that it can extract an object reference from an invocation request, and subsequently dispatch the request to the referenced object, but now following a specific activation policy. The stub, also called a skeleton, is normally generated from the interface definitions of the object, unmarshals the request and invokes the appropriate method. An object adapter can support different activation policies by simply configuring it at runtime. For example, in CORBA-compliant systems it is possible to specify whether an object should continue to exist after its associated adapter has stopped. Likewise, an adapter can be configured to generate object identifiers, or to let the application provide one. As a final example, an adapter can be configured to operate in single-threaded ormultithreaded mode as we explained above. In particular, it should be stressed that as part of the implementation of such an object the server may (indirectly) access databases or call special library routines. The implementation details are hidden for the object adapter who communicates only with a skeleton. As such, the actual implementation may have nothing to do with what we often see with language-level (i.e., compile-time) objects. For this reason, a different terminology is generally adopted. A **servant** is the general term for a piece of code that forms the implementation of an object.

Apache Web server

An interesting example of a server that balances the separation between policies and mechanisms is the Apache Web server. It is also an extremely popular server, estimated to be used to host approximately 50% of all Web sites. Apache is a complex piece of software, and with the numerous enhancements to the types of documents that are now offered in the Web, it is important that the server is highly configurable and extensible, and at the same time largely independent of specific platforms.

Making the server platform independent is realized by essentially providing its own basic runtime environment, which is then subsequently implemented for different operating systems. This runtime environment, known as the Apache Portable Runtime (APR), is a library that provides a platform- independent interface for file handling, networking, locking, threads, and so on. When extending Apache, portability is largely guaranteed provided that only calls to the APR are made and that calls to platform-specific libraries are avoided.

From a certain perspective, Apache can be considered as a completely gen- eral server tailored to produce a response to an incoming request. Of course, there are all kinds of hidden dependencies and assumptions by which Apache turns out to be primarily suited for handling requests for Web documents. For example, as we mentioned, Web browsers and servers use HTTP as their communication protocol. HTTP is virtually always implemented on top of TCP, for which reason the core of Apache assumes that all incoming requests adhere to a TCP-based connection-oriented way of communication. Requests based on UDP cannot be handled without modifying the Apache core.

However, the Apache core makes few assumptions on how incoming requests should be handled. Fundamental to this organization is the concept of a **hook**, which is nothing but a placeholder for a specific group of functions. The Apache core assumes that requests are processed in a number of phases, each phase consisting of a few hooks. Each hook thus represents a group of similar actions that need to be executed as part of processing a request.

For example, there is a hook to translate a URL to a local file name. Such a translation will almost certainly need to be done when processing a request. Likewise, there is a hook for writing information to a log, a hook for checking a client's identification, a hook for checking access rights, and a hook for checking which MIME type the request is related to (e.g., to make sure that the request can be properly handled). The hooks are processed in a predetermined order. It is here that we explicitly see that Apache enforces a specific flow of control concerning the processing of requests.

The functions associated with a hook are all provided by separate **modules**. Although, in principle, a developer could change the set of hooks that will be processed by Apache, it is far more common to write modules containing the functions that need to be called as part of processing the standard hooks provided by unmodified Apache. The underlying principle is fairly straightforward. Every hook can contain a set of functions that each should match a specific function prototype. A module developer will write functions for specific hooks. When compiling Apache, the developer specifies which function should be added to which hook. The latter is shown in Figure 3.17 as the various links between functions and hooks because there may be tens of modules, each hook will generally contain several functions. Normally, modules are considered to be mutual independent, so that functions in the same hook will be executed in some arbitrary order. However, Apache can also handle module dependencies by letting a developer specify an ordering in which functions from different modules should be processed. By and large, the result is a Web server that is extremely versatile.

Server clusters

We briefly discussed cluster computing as one of the many appearances of distributed systems. We now take a closer look at the organization of server clusters, along with the salient design issues. We first consider common server clusters that are organized in local-area networks. A special group is formed by wide-area server clusters, which we subsequently discuss.

Local Aarea Clusters

Simply put, a server cluster is nothing else but a collection of machines connected through a network, where each machine runs one or more servers. The server clusters that we consider here, are the ones in which the machines are connected through a local-area network, often offering high bandwidth and low latency.

General Organization

In many cases, a server cluster is logically organized into three tiers. The first tier consists of a (logical) switch through which client requests are routed. Such a switch can vary widely. For example, transport-layer switches accept incoming TCP connection requests and pass requests on to one of servers in the cluster. A completely different example is a Web server that accepts incoming HTTP requests, but that partly passes requests to application servers for further processing only to later collect results from those servers and return an HTTP response.

As in any multitier client-server architecture, many server clusters also contain servers dedicated to application processing. In cluster computing, these are typically servers running on high-performance hardware dedicated to delivering compute power. However, in the case of enterprise server clusters, it may be the case that applications need only run on relatively low-end machines, as the required compute power is not the bottleneck, but access to storage.

This brings us the third tier, which consists of data-processing servers, notably file and database servers. Again, depending on the usage of the servercluster, these servers may be running on specialized machines, configured for high-speed disk access and having large server-side data caches. Of course, not all server clusters will follow this strict separation. It is frequently the case that each machine is equipped with its own local storage, often integrating application and data processing in a single server leading to a two-tiered architecture. For example, when dealing with streaming mediaby means of a server cluster, it is common to deploy a two-tiered system architecture, where each machine acts as a dedicated media server.

When a server cluster offers multiple services, it may happen that different machines run different application servers. As a consequence, the switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines. As a consequence, we may find that certain machines are temporarily idle, while others are receiving an overload of requests. What would be useful is to temporarily migrate services to idle machines. A solution is to use virtual machines allowing a relatively easy migration of code to real machines.

Request Dispatching

Let us now take a closer look at the first tier, consisting of the switch, also known as the front end. An important design goal for server clusters is to hide the fact that there are multiple servers. In other words, client applications running on remote machines should have no need to know anything about the internal organization of the cluster. This access transparency is invariably offered by means of a single access point, in turn implemented through some kind of hardware switch such as a dedicated machine. The switch forms the entry point for the server cluster, offering a single network address. For scalability and availability, a server cluster may have multiple access points, where each access point is then realized by a separatededicated machine. We consider only the case of a single access point.

A standard way of accessing a server cluster is to set up a TCP connection over which application-level requests are then sent as part of a session as display. A session ends by tearing down the connection. In the case of transport-layer switches, the switch accepts incoming TCP connection requests, and hands off such connections to one of the servers. There are essentially two ways how the switch can operate. In the first case, the client sets up a TCP connection such that all requests and responses pass through the switch. The switch, in turn, will set up a TCP connection with a selected server and pass client requests to that server,

and also accept server responses. In effect,the switch sits in the middle of a TCP connection between the client and a selected server, rewriting the source and destination addresses when passing TCP segments. This approach is a form of network address translation (NAT).

When the switch receives a TCP connection request, it first identifies thebest server for handling that request, and forwards the request packet to that server. The server, in turn, will send an acknowledgment back to the requesting client, but inserting the switch's IP address as the source field of the header of the IP packet carrying the TCP segment. Note that this addressrewriting is necessary for the client to continue executing the TCP protocol: it is expecting an answer back from the switch, not from some arbitrary server it has never heard of before. Clearly, a TCP-handoff implementation requires operating-system level modifications. TCP handoff is especially effective when responses are much larger than requests, as in the case of Web servers.

It can already be seen that the switch can play an important role in distributing the load among the various servers. By deciding where to forward a request to, the switch also decides which server is to handle further processing of the request. The simplest load-balancing policy that the switch can followis round robin: each time it picks the next server from its list to forward a request to. Of course, the switch will have to keep track to which server it handed off a TCP connection, at least until that connection is torn down. As it turns out, maintaining this state and handing off subsequent TCP segments belonging to the same TCP connection, may actually slow down the switch.

More advanced server selection criteria can be deployed as well. For example, assume multiple services are offered by the server cluster. If the switch can distinguish those services when a request comes in, it can then take informed decisions on where to forward the request to. This server selection can still take place at the transport level, provided services are distinguished by means of a port number. In the case of transport-level switches, as we have discussed so far, decisions on where to forward an incoming request is based on transport-level information only. One step further is to have the switch actually inspect the payload of the incoming request. This content-aware request distribution can be applied only if it is known what that payload looks like. For example, in the case of Web servers, the switch can expect an HTTP request, based on which it can then decide who is to process it.

CHAPTER 15

WIDE AREA CLUSTERS

Gaurav Kumar Rajput, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- gauravrajput31@gmail.com

A characteristic feature of local-area server clusters is that they are ownedby a single organization. Deploying clusters across a wide-area network has traditionally been quite cumbersome as one had to generally deal with multiple administrative organizations such as ISPs (Internet Service Providers). With the advent of cloud computing, matters have changed and we are now witnessing an increase of wide-area distributed systems in which servers or server clusters are spread across the Internet. The problems related to having to deal with multiple organizations are effectively circumvented by making use of the facilities of a single cloud provider.

Cloud providers like Amazon and Google manage several data centers placed at different locations worldwide. As such, they can offer an end user the ability to build a wide-area distributed system consisting of a potentially large collection of networked virtual machines, scattered across the Internet. An important reason for wanting such distributed systems is to provide locality: offering data and services that are close to clients. An example where such locality is important is streaming media: the closer a video server is located to a client, the easier it becomes to provide high-quality streams. Note that if wide-area locality is not critical, it may suffice, or even be better, to place virtual machines in a single data center, so that intercrosses communication can benefit from low-latency local networks. The price to pay may be higher latencies between clients and the service running in a remote data center.

Request dispatching: If wide-area locality is an issue, then request dispatching becomes important: if a client accesses a service, its request should be forwarded to a nearby server, that is, a server that will allow communication with that client to be fast. Deciding which server should handle the client's request is an issue of **redirection policy:** If we assume that a client will initially contact a request dispatcher analogous to the switch in our discussion of local-area clusters, then that dispatcher will have to estimate the latency between the client and several servers. How such an estimation can be made is discussed in Section 6.5.

Once a server has been selected, the dispatcher will have to inform the client. Several **redirection mechanisms** are possible. A popular one is when the dispatcher is actually a DNS name server. Internet or Web-based services are often looked up in the **Domain Name System** (**DNS**). A client provides a domain name such as service.organization.org to a local DNS server, which eventually returns an IP address of the associated service, possibly after having contacted other DNS servers. When sending its request to look up a name, a client also sends its own IP address (DNS requests are sent as UDP packets). In other words, the DNS server will also know the client's IP address which it can then subsequently use to select the best server for that client, and returning a close-by IP address.

Unfortunately, this scheme is not perfect for two reasons. First, rather than sending the client's IP address, what happens is that the local DNS server that is contacted by the client acts as a proxy for that client. In other words, not the client's IP address, but that of the local DNS

server is used to identify the location of the cliente and have shown that there may be a huge additional communication cost, as the local DNS server is often not *that* local. Secondly, depending on the scheme that is used for resolving a domain name, it may even be the case that the address of the local DNS server is not even being used.

Instead, it may happen that the DNS server that is deciding on which IP address to return, may be fooled by the fact that the requester is yet another DNS server acting as an intermediate between the original client and the deciding DNS server. In those case, locality awareness has been completely lost. Despite that DNS-based redirection may not always be very accurate, it is widely deployed if only for the fact that it is relatively easy to implement and also transparent to the client. In addition, there is no need to rely on location-aware client-side software.

Code Migration

So far, we have been mainly concerned with distributed systems in which communication is limited to passing data. However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system. In this section, we take a detailed look at what code migration actually is. We start by considering different approaches to code migration, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems, which is also discussed.

Reasons for migrating code

Traditionally, code migration in distributed systems took place in the form of **process migration** in which an entire process was moved from one node to another. Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily loaded to lightly loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well. When completing their survey, had already come to the conclusion that process migration was no longer a viable option for improving distributed systems.

However, instead of offloading machines, we can now witness that code is moved to make sure that a machine is sufficiently loaded. In particular, migrating complete virtual machines with their suite of applications to lightly loaded machines in order to minimize the total number of nodes being used is common practice in optimizing energy usage in data centers. In general, load-distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of machines, play an important role in compute-intensive systems. However, in many modern distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, as an example, a client-server system in which the server man- ages a huge database. If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill informs that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication. In the case of smartphones, moving code to be executed at the handheld instead of the server may be the only viable solution to obtain acceptable performance, *both* for the client and the server.

Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance. However, conclude that mobile agents have never become successful because they did not really offer an obvious advantage over other technologies. Moreover, and crucial, it turned out to be virtually impossible to let this type of mobile code operate in a secure way. Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application.This approach requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Typically, scripts that run in a virtual machine embedded in, for example, a Web browser, will do the trick. Arguably, this form of code migration has been key to the success of the dynamic Web. These and other solutions are discussed below and in laterchapters.

The important advantage of this model of dynamically downloading client- side software is that clients need not have all the software preinstalled to talk to servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server. There are, of course, also disadvantages. Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven- knows-who may not always be such a good idea. Fortunately, it is well understood how to protect the client against malicious, downloaded code.

CHAPTER 16

MIGRATION IN HETEROGENEOUS SYSTEMS

Aaditya Jain, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- jain.aaditya58@gmail.com

So far, we have tacitly assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous systems. In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture.

The problems coming from heterogeneity are in many respects the same as those of portability. Not surprisingly, solutions are also very similar. For example, at the end of the 1970s, a simple solution to alleviate many of the problems of porting Pascal to different machines was to generate machine- independent intermediate code for an abstract virtual machine. That machine, of course, would need to be implemented on many platforms, but it would then allow Pascal programs to be run anywhere. Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C.

About 25 years later, code migration in heterogeneous systems is being tackled by scripting languages and highly portable languages such as Java. In essence, these solutions adopt the same approach as was done for porting Pascal. All such solutions have in common that they rely on a (process) virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

Further developments have weakened the dependency on programming languages. In particular, solutions have been proposed to migrate not only processes, but to migrate entire computing environments. The basic idea is to compartmentalize the overall environment and to provide processes in the same part their own view on their computing environment. That compartmentalization takes place in the form of virtual machine monitors running an operating system and a suite of applications.

With virtual machine migration, it becomes possible to decouple a computing environment from the underlying system and actually migrate it to another machine. A major advantage of this approach is that processes can remain ignorant of the migration itself: they need not be interrupted in their execution, nor should they experience any problems with used resources. The latter are either migrating along with a process, or the way that a process accesses a resource is left unaffected at least, for that process.

As an example, concentrated on real-time migration of a virtualized operating system, typically something that would be convenient in a cluster of servers where a tight coupling is achieved through a single,shared local-area network. Under these circumstances, migration involves two major problems: migrating the entire memory image and migrating bindings to local resources.

As to the first problem, there are, in principle, three ways to handle migration which can be combined:

i. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.

ii. Stopping the current virtual machine; migrate memory, and start the new virtual machine.

iii. Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.

The second option may lead to unacceptable downtime if the migrating virtual machine is running a live service, that is, one that offers continuous service. On the other hand, a pure on-demand approach as represented by the third option may extensively prolong the migration period, but may also lead to poor performance because it takes a long time before the working setof the migrated processes has been moved to the new machine.

As an alternative, propose to use a pre-copy approach which combines the first option, along with a brief stop-and-copy phase as represented by the second option. As it turns out, this combination can lead to very low service downtimes. Concerning local resources, matters are simplified when dealing only with a cluster server. First, because there is a single network, the only thing that needs to be done is to announce the new network-to-MAC address binding, so that clients can contact the migrated processes at the correct network interface.

In many cases, virtual machines are migrated to optimize the usage of actual machines. However, it may also be desirable to clone a virtual machine, for example, because the workload for the current machine is becoming too high. Such cloning is very similar to using multiple processes in concurrent servers bywhich a dispatcher process creates worker processes to handle incoming requests. This scheme was explained in Figure 3.4 when discussing multithreaded servers. When cloning for this type of performance, it often makes more sense *not* to first copy memory pages, but, in fact, start with as few pages as possible as the service running on the cloned machine will essentially start anew. Note that this behavior is very similar to the usual parent-child behavior we see when forking Unix process. Namely, the child will start with loading its own executable, thereby effectively cleaning the memory it inherited from its parent. This analogy inspired to develop an analogous mechanism for *forking* a virtual machine. However, unlike the mechanism used traditionally for migrating virtual machines, their VM fork copies pages primarily on demand. The result is an extremely efficient cloning mechanism.

It is thus seen that there is no single best way to place copies of a virtual machine on different physical machines: it very much depends on how and why a virtual machine is being deployed special attention needs to be paid when organizing servers into a cluster. A common objective is to hide the internals of a cluster from the outside world. This means that the organization of the cluster should be shielded from applications. To this end, most clusters use a single entry point that can hand off messages to servers in the cluster. A challenging problem is to transparently replace this single entry point by a fully distributed solution.

Advanced object servers have been developed for hosting remote objects. An object server provides many services to basic objects, including facilities for storing objects, or to ensure serialization of incoming requests. Another important role is providing the illusion to the

outside world that a collection of data and procedures operating on that data correspond to the concept of an object. This role is implemented by means of object adapters. Object-based systems have come to a point where we can build entire frameworks that can be extended for supporting specific applications. Java has proven to provide a powerful means for setting up more generic services, exemplified by the highly popular Enterprise Java Beans concept and its implementation.

An exemplary server for Web-based systems is the one from Apache. Again, the Apache server can be seen as a general solution for handling a myriad of HTTP-based queries. By offering the right hooks, we essentially obtain a flexibly configurable Web server. Apache has served as an example not only for traditional Web sites, but also for setting up clusters of collaborative Web servers, even across wide-area networks.

An important topic for distributed systems is the migration of code be- tween different machines. Two important reasons to support code migration are increasing performance and flexibility. When communication is expensive, we can sometimes reduce communication by shipping computations from the server to the client, and let the client do as much local processing as possible. Flexibility is increased if a client can dynamically download software needed to communicate with a specific server. The downloaded software can be specifically targeted to that server, without forcing the client to have it preinstalled.

Code migration brings along problems related to usage of local resources for which it is required that either resources are migrated as well, new bindings to local resources at the target machine are established, or for which system wide network references are used. Another problem is that code migration requires that we take heterogeneity into account. Current practice indicates that the best solution to handle heterogeneity is to use virtual ma- chines. These can take either the form of process virtual machines as in the case of, for example, Java, or through using virtual machine monitors that effectively allow the migration of a collection of processes along with their underlying operating system.

CHAPTER 17

AN OVER VIEW OF THE DISTRIBUTED COMMUNICATION

Ashendra Kumar Saxena, Professor & Vice Principal
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- ashendrasaxena@gmail.com

Interposes communication is at the heart of all distributed systems. It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information. Communication in distributed systems has traditionally always been based on low-levelmessage passing as offered by the underlying network. Expressing communication through message passing is harder than using primitives based on shared memory, as available for no distributed platforms. Modern distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet. Unlessthe primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

In this chapter, we start by discussing the rules that communicating processes must adhere to, known as protocols, and concentrate on structuring those protocols in the form of layers. We then look at two widely-used models for communication: Remote Procedure Call (RPC), and Message-Oriented Middleware (MOM). We also discuss the general problem of sending data to multiple receivers, called multicasting.

Our first model for communication in distributed systems is the remote procedure call (RPC). An RPC aims at hiding most of the intricacies of message passing, and is ideal for client-server applications. However, realizing RPCs in a transparent manner is easier said than done. We look at a number of important details that cannot be ignored, while diving into actually code to illustrate to what extent distribution transparency can be realized such thatperformance is still acceptable.

In many distributed applications, communication does not follow the rather strict pattern of client-server interaction. In those cases, it turns out that thinking in terms of messages is more appropriate. The low-level communication facilities of computer networks are in many ways not suitable, again due to their lack of distribution transparency. An alternative is to use ahigh-level message-queuing model, in which communication proceeds much the same as in e-mail systems. Message-oriented communication is a subjectimportant enough to warrant a section of its own. We look at numerous aspects, including application-level routing.

Finally, since our understanding of setting up multicast facilities has im- proved, novel and elegant solutions for data dissemination have emerged. We pay separate attention to this subject in the last section of this chapter, discussing traditional deterministic means of multicasting, as well as proba- bilistic approaches as used in flooding and gossiping. The latter have been receiving increased attention over the past years due to their elegance and simplicity.

Layered Protocols

Due to the absence of shared memory, all communication in distributed systems is based on sending and receiving low level messages. When process P wants to communicate with process Q, it first builds a message in its own address space. Then it executes a system call that causes the operating system to send the message over the network to Q. Although this basic idea sounds simple enough, in order to prevent chaos, P and Q have to agree on the meaning of the bits being sent[21].

The OSI reference model

To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job. This model is called the Open Systems Interconnection Reference Model usually abbreviated as **ISO OSI** or sometimes just the **OSI model**. It should be emphasized that the protocols that were developed as part of the OSI model were never widely used and are essentially dead. However, the underlying model itself has proved to be quite useful for understanding computer networks. Although we do not intend to give a full description of this model and all of its implications here, a short introduction will be helpful.

The OSI model is designed to allow open systems to communicate. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called communication protocols. To allow a group of computers to communicate over a network, they must all agree on the protocols to be used. A protocol is said to provide a communication service. There are two types of such services. In the case of a connection-oriented service, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate specific parameters of the protocol they will use. When they are done, they release (terminate) the connection. The telephone is a typical connection-oriented communication service. With connectionless services, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of making use of connectionless communication service. With computers, both connection-oriented and connectionless communication are common.

In the OSI model, communication is divided into seven levels or layers. Each layer offers one or more specific communication services to the layer above it. In this way, the problem of getting a message from A to B can be divided into manageable pieces, each of which can be solved independently of the others. Each layer provides an interface to the one above it. The interface consists of a set of operations that together define the service the layer is prepared to offer. The seven OSI layers are:

- **Physical layer** Deals with standardizing how two computers are connected and how 0s and 1s are represented.
- **Data link layer** provides the means to detect and possibly correct transmission errors, as well as protocols to keep a sender and receiver in the same pace.
- **Network layer** Contains the protocols for routing a message through a computer network, as well as protocols for handling congestion.

- **Transport layer** mainly contains protocols for directly supporting applications, such as those that establish reliable communication, or support real-time streaming of data.
- **Session layer** Provides support for sessions between applications.
- **Presentation layer** prescribes how data is represented in a way that is inde- pendent of the hosts on which communicating applications are running.
- **Application layer** essentially, everything else: e-mail protocols, Web access protocols, file-transfer protocols, and so on.

When process P wants to communicate with some remote process Q, it builds a message and passes that message to the application layer as offered to it by means of an interface. This interface will typically appear in the form of a library procedure. The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer, in turn, adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits the bottom, the physical layer actually transmits the message by putting it onto the physical transmission medium. When the message arrives at the remote machine hosting Q, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process Q, which may reply to it using the reverse path. The information in the layer-n header is used for the layer-n protocol.

In the OSI model, there are not two layers, but seven, as we saw in Figure 41. The collection of protocols used in a particular system is called a protocol suite or protocol stack. It is important to distinguish a reference model from its actual protocols. As said, the OSI protocols were never popular, in contrast to protocols developed for the Internet, such as TCP and IP.

Middleware Protocols

Middleware is an application that logically lives (mostly) in the OSI application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications. Let us briefly look at some examples.

The Domain Name System (DNS) [Liu and Albitz, 2006] is a distributed service that is used to look up a network address associated with a name, such as the address of a so-called domain name like www.distributed-systems.net. In terms of the OSI reference model, DNS is an application and therefore is logically placed in the application layer. However, it should be quite obvious that DNS is offering a general-purpose, application-independentservice. Arguably, it forms part of the middleware.

As another example, there are various ways to establish authentication, that is, provide proof of a claimed identity. Authentication protocols are not closely tied to any specific application, but instead, can be integrated into a middleware system as a general service. Likewise, authorization protocols by which authenticated users and processes are granted access only to those resources for which they have authorization, tend to have a general, application-independent nature. Being labeled as applications in the OSI reference model, these are clear examples that belong in the middleware.

Distributed commit protocols establish that in a group of processes, possibly spread out across a number of machines, either all processes carry out a particular operation, or that

the operation is not carried out at all. This phenomenon is also referred to as **atomicity** and is widely applied in transactions. As it turns out, commit protocols can present an interface independently of specific applications, thus providing a general-purpose transaction service.

In such a form, they typically belong to the middleware and not to the OSI application layer. As a last example, consider a distributed locking protocol by which a resource can be protected against simultaneous access by a collection of processes that are distributed across multiple machines. It is not hard to imagine that such protocols can be designed in an application-independent fashion, and accessible through a relatively simple, again application-independent interface. As such, they generally belong in the middleware.

These protocol examples are not directly tied to communication, yet there are also many middleware communication protocols. For example, with a so-called remote procedure call, a process is offered a facility to locally call a procedure that is effectively implemented on a remote machine. This communication service belongs to one of the oldest types of middleware services and is used for realizing access transparency. In a similar vein, there are high-level communication services for setting and synchronizing streams for transferring real-time data, such as needed for multimedia applications. As a last example, some middleware systems offer reliable multicast services that scale to thousands of receivers spread across a wide-area network.

Taking this approach to layering leads to the adapted and simplified reference model for communication. Compared to the OSI model, the session and presentation layer have been replaced by a single middleware layer that contains application-independent protocols. These protocols do not belong in the lower layers we just discussed. Network and transport services have been grouped into communication services as normally offered by an operating system, which, in turn, manages the specific lowest-level hardware used to establish communication.

Types of Communication

In the remainder of this chapter, we concentrate on high-level middleware communication services. Before doing so, there are other general criteria for distinguishing (middleware) communication. To understand the various alternatives in communication that middleware can offer to applications, we view the middleware as an additional service in client-server computing, as shown in Figure 3. Consider, for example an electronic mail system. In principle, the core of the mail delivery system can be seen as a middleware communication service. Each host runs a user agent allowing users to com- pose, send, and receive e-mail. A sending user agent passes such mail to the mail delivery system, expecting it, in turn, to eventually deliver the mail to the intended recipient. Likewise, the user agent at the receiver's side connects to the mail delivery system to see whether any mail has come in. If so, the messages are transferred to the user agent so that they can be displayed and read by the user [22].

An electronic mail system is a typical example in which communication is persistent. With persistent communication, a message that has been sub- mitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver. In this case, the middleware will store the message at one or several of the storage facilities. As a consequence, it is not necessary for the sending application to continue execution after submitting the message. Likewise, the receiving application need not be executing when the message is submitted.

In contrast, with transient communication, a message is stored by the communication system only as long as the sending and receiving application are executing. More precisely, in terms of if the middleware cannot deliver a message due to a transmission interrupt, or because the recipient is currently not active, it will simply be discarded. Typically, all transport-level communication services offer only transient communication. In this case, the communication system consists of traditional store-and-forward routers. If a router cannot deliver a message to the next one or the destination host, it will simply drop the message.

Besides being persistent or transient, communication can also be asynchronous or synchronous. The characteristic feature of asynchronous communication is that a sender continues immediately after it has submitted its message for transmission. This means that the message is (temporarily) stored immediately by the middleware upon submission. With synchronous communication, the sender is blocked until its request is known to be accepted. There are essentially three points where synchronization can take place. First, the sender may be blocked until the middleware notifies that it will take over transmission of the request. Second, the sender may synchronize until its request has been delivered to the intended recipient. Third, synchronization may take place by letting the sender wait until its request has been fully processed, that is, up to the time that the recipient returns a response.

Various combinations of persistence and synchronization occur in practice. Popular ones are persistence in combination with synchronization at request submission, which is a common scheme for many message-queuing systems, which we discuss later in this chapter. Likewise, transient communication with synchronization after the request has been fully processed is also widely used. This scheme corresponds with remote procedure calls, which we discuss next.

Remote Procedure Call

Many distributed systems have been based on explicit message exchange between processes. However, the operations send and receive do not conceal communication at all, which is important to achieve access transparency in distributed systems. This problem has long been known, but little was done about it until researchers in the 1980s introduced a completely different way of handling communication. Although the idea is refreshingly simple (once someone has thought of it), the implications areoften subtle. In this section we will examine the concept, its implementation,its strengths, and its weaknesses.

In a nutshell, the proposal was to allow programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer.

While the basic idea sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different ma- chines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, either or both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

Basic RPC operation

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be

aware that the called procedure is executingon a different machine or vice versa. Suppose that a program has access to a database that allows it to append data to a stored list, after which it returns a reference to the modified list. The operation is made available to a programby means of a routine append:

newlist = append(data, dbList)

In a traditional (single-processor) system, append is extracted from a libraryby the linker and inserted into the object program. In principle, it can be a short procedure, which could be implemented by a few file operations for accessing the database.

Even though append eventually does only a few basic file operations, it is called in the usual way, by pushing its parameters onto the stack. The programmer does not know the implementation details of append, and this is,of course, how it is supposed to be.

Language-based Support

The approach described up until now is largely independent of a specific programming language. As an alternative, we can also embed remote procedure calling into a language itself. The main benefit is that application development often becomes much simpler. Also, reaching a high degree of access transparency is often simpler as many issues related to parameter passing can be circumvented altogether.

A well-known example in which remote procedure calling is fully embedded is Java, where an RPC is referred to as a remote method invocation(RMI). In essence, a client being executed by its own (Java) virtual machine can invoke a method of an object managed by another virtual machine. By simply reading an application's source code, it may be hard or even impossible to see whether a method invocation is to a local or to a remote object.

Multicast RPC

Asynchronous and deferred synchronous RPCs facilitate another alternative to remote procedure calls, namely executing multiple RPCs at the same time. Adopting the one-way RPCs (i.e., when a server does not tell the client it has accepted its call request but immediately starts processing it), a multicast RPC boils down to sending an RPC request to a group of servers. In this example, the client sends a request to two servers, who subsequently process that request independently and in parallel. When done, the result is returned to the client where a callback takes place.

There are several issues that we need to consider. First, as before, the client application may be unaware of the fact that an RPC is actually being forwarded to more than one server. For example, to increase fault tolerance, we may decide to have all operations executed by a backup server who can take over when the main server fails. That a server has been replicated can be completely hidden from a client application by an appropriate stub. Yet even the stub need not be aware that the server is replicated, for example, because we are using a transport-level multicast address[23].

Second, we need to consider what to do with the responses. In particular, will the client proceed after all responses have been received, or wait just forone? It all depends. When the server has been replicated for fault tolerance, we may decide to wait for just the first response, or perhaps until a majorityof the servers returns the same result. On the other hand, if the servers havebeen replicated to do the same work but on different parts of the input, theirresults may need to be merged before the client can continue. Again, such

matters can be hidden in the client-side stub, yet the application developer will, at the very least, have to specify the purpose of the multicast RPC.

Example: DCE RPC

Remote procedure calls have been widely adopted as the basis of middleware and distributed systems in general. In this section, we take a closer look at the Distributed Computing Environment (DCE) which was developed by theOpen Software Foundation (OSF), now called The Open Group. It forms the basis for Microsoft's distributed computing environment DCOM and used in Samba, a file server and accompanying protocol suite allowing the Windows file system to be accessed through remote procedurecalls from non-Windows systems.

Although DCE RPC is arguably not the most modern way of managing RPCs, it is worthwhile discussing some of its details, notably because itis representative for most traditional RPC systems that use a combination of interface specifications and explicit bindings to various programminglanguages. We start with a brief introduction to DCE, after which we consider its principal workings. Details on how to develop RPC-based applications canbe found.

CHAPTER 18

INTRODUCTION TO DCE

Rajendra P. Pandey, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- panday_004@yahoo.co.uk

DCE is a true middleware system in that it is designed to execute as a layer of abstraction between existing (network) operating systems and distributed applications. Initially designed for Unix, it has now been ported to all major operating systems. The idea is that the customer can take a collection of existing machines, add the DCE software, and then be able to run distributed applications, all without disturbing existing applications.

Although most of the DCE package runs in user space, in some configurations a piece must be added to the kernel of the underlying operating system. The programming model underlying DCE is the client-server model. User processes act as clients to access remote services provided by server processes. Some of these services are part of DCE itself, but others belong to the applications and are written by the application programmers. All communication between clients and servers takes place by means of RPCs.

Goals of DCE RPC

The goals of the DCE RPC system are relatively traditional. First and foremost, the RPC system makes it possible for a client to access a remote service by simply calling a local procedure. This interface makes it possible for client that is application programs to be written in a simple way, familiar to most programmers. It also makes it easy to have large volumes of existing code run in a distributed environment with few, if any, changes. It is up to the RPC system to hide all the details from the clients, and, to some extent, from the servers as well. To start with, the RPC system can automatically locate the correct server, and subsequently set up the communication between client and server software. It can also handle the message transport in both directions, fragmenting and reassembling them as needed. Finally, the RPC system can automatically handle data type conversions between the client and the server, even if they run on different architectures and have a different byte ordering[24].

As a consequence of the RPC system's ability to hide the details, clients and servers are highly independent of one another. A client can be written in Java and a server in C, or vice versa. A client and server can run on different hardware platforms and use different operating systems. A variety of network protocols and data representations are also supported, all without any intervention from the client or server.

Writing a Client and a Server

The DCE RPC system consists of a number of components, including languages, libraries, daemons, and utility programs, among others. Together these make it possible to write clients and servers. In this section we will describe the pieces and how they fit together.

In a client-server system, the glue that holds everything together is the interface definition, as specified in the **Interface Definition Language**, or **IDL**. It permits procedure declarations in a form closely resembling function prototypes in ANSI C. IDL files can also

contain type definitions, constant declarations, and other information needed to correctly marshal parameters and unmarshal results. Ideally, the interface definition should also contain a formal definition of what the procedures do, but such a definition is beyond the current state of the art, so the interface definition just defines the syntax of the calls, not their semantics. At best the writer can add a few comments describing what the procedures do.

A crucial element in every IDL file is a globally unique identifier for the specified interface. The client sends this identifier in the first RPC message and the server verifies that it is correct. In this way, if a client inadvertently tries to bind to the wrong server, or even to an older version of the right server, the server will detect the error and the binding will not take place. Interface definitions and unique identifiers are closely related in DCE. The first step in writing a client/server application is usually calling the uuidgen program, asking it to generate a prototype IDL file containing an interface identifier guaranteed never to be used again in any interface generated anywhere by uuidgen. Uniqueness is ensured by encoding in it the location and time of creation. It consists of a 128-bit binary number represented in the IDL file as an ASCII string in hexadecimal.

The next step is editing the IDL file, filling in the names of the remote procedures and their parameters. It is worth noting that RPC is not totally transparent. For example, the client and server cannot share global variables. The IDL rules make it impossible to express constructs that are not supported. When the IDL file is complete, the IDL compiler is called to process it. The output of the IDL compiler consists of three files:

- **a.** A header file (e.g., interface.h, in C terms).
- **b.** The client stub.
- **c.** The server stub.

The header file contains the unique identifier, type definitions, constant definitions, and function prototypes. It should be included (using #include) in both the client and server code. The client stub contains the actual procedures that the client program will call. These procedures are the ones responsible for collecting and packing the parameters into the outgoing message and then calling the runtime system to send it. The client stub also handles unpacking the reply and returning values to the client. The server stub contains the procedures called by the runtime system on the server machine when an incoming message arrives. These, in turn, call the actual server procedures that do the work.

The next step is for the application writer to write the client and server code. Both of these are then compiled, as are the two stub procedures. The resulting client code and client stub object files are then linked with the runtime library to produce the executable binary for the client. Similarly, the server code and server stub are compiled and linked to produce the server's binary. At runtime, the client and server are started so that the application is actually executed as well.

Binding a client to a server

To allow a client to call a server, it is necessary that the server has been registered and is prepared to accept incoming calls. Registration of a server makes it possible for a client to locate the server and bind to it. Finding the location of the server is done in two steps:

- **a.** Locate the server's machine.
- **b.** Locate the server (i.e., the correct process) on that machine.

The second step is somewhat subtle. Basically, what it comes down to is that to communicate with a server, the client needs to know a port on the server's machine to which it can send messages. A port is used by the server's operating system to distinguish incoming messages for different processes. In DCE, a table of (server, port) pairs is maintained on each server machine bya process called the DCE daemon. Before it becomes available for incoming requests, the server must ask the operating system for a port. It then registers this port with the DCE daemon. The DCE daemon records this information including which protocols the server speaks in the port table for future use. The server also registers with the directory service by providing it the network address of the server's machine and a name under which the servercan be looked up. Binding a client to a server proceeds.

Let us assume that the client wants to bind to a video server that is locally known under the name /local/multimedia/video/movies. It passes this name to the directory server, which returns the network address of the machinerunning the video server. The client then goes to the DCE daemon on that machine which has a well-known port, and asks it to look up the port of the video server in its port table. Armed with this information, the RPC can now take place. On subsequent RPCs this lookup is not needed. DCE also gives clients the ability to do more sophisticated searches for a suitable server when that is needed. Secure RPC is also an option where confidentiality or data integrity is crucial.

Performing an RPC

The actual RPC is carried out transparently and in the usual way. The client stub marshals the parameters to the runtime library for transmission using the protocol chosen at binding time. When a message arrives at the server side, itis routed to the correct server based on the port contained in the incoming message. The runtime library passes the message to the server stub, which marshals the parameters and calls the server. The reply goes back by the reverse route.

DCE provides several semantic options. The default is at-most-once operation, in which case no call is ever carried out more than once, even in the presence of system crashes. In practice, what this means is that if a server crashes during an RPC and then recovers quickly, the client does not repeat the operation, for fear that it might already have been carried out once.

Alternatively, it is possible to mark a remote procedure as idempotent in the IDL file, in which case it can be repeated multiple times without harm. For example, reading a specified block from a file can be tried over and over until it succeeds. When an idempotent RPC fails due to a server crash, the client can wait until the server reboots and then try again. Other semantics are also available but rarely used, including broadcasting the RPC to all the machines on the local network.

Message Oriented Communication

Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency. Unfortunately, neither mechanism is always appropriate. In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed. Likewise, the inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, may need to be replaced by something else.

That something else is messaging. In this section we concentrate on message-oriented communication in distributed systems by first taking a closer look at what exactly synchronous behavior is and what its implications are. Then, we discuss messaging

systems that assume that parties are executing at the time of communication. Finally, we will examine message-queuing systems that allow processes to exchange information, even if the other party is not executing at the time communication is initiated.

Simple Transient Messaging with Sockets

Many distributed systems and applications are built directly on top of the simple message-oriented model offered by the transport layer. To better understand and appreciate the message-oriented systems as part of middleware solutions, we first discuss messaging through transport-level sockets. Special attention has been paid to standardizing the interface of the trans- port layer to allow programmers to make use of its entire suite of protocols through a simple set of operations. Also, standard interfaces make it easier to port an application to a different machine. As an example, we briefly discuss the socket interface as introduced in the 1970s in Berkeley Unix, and which has been adopted as a POSIX standard. Conceptually, a socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual port that is used by the local operating system for a specific transport protocol.

Servers generally execute the first four operations, normally in the order given. When calling the socket operation, the caller creates a new communication end point for a specific transport protocol. Internally, creating a communication end point means that the local operating system reserves resources for sending and receiving messages for the specified protocol. The bindoperation associates a local address with the newly created socket. For example, a server should bind the IP address of its machine together with a port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port. In the case of connection-oriented communication, the address isused to receive incoming connection requests [25].

The listen operation is called only in the case of connection-oriented communication. It is a non-blocking call that allows the local operating sys- tem to reserve enough buffers for a specified maximum number of pending connection requests that the caller is willing to accept. A call to acceptblocks the caller until a connection request arrives. Whena request arrives, the local operating system creates a new socket with the same properties as the original one, and returns it to the caller. This approach will allow the server to, for example, fork off a process that will subsequently handle the actual communication through the new connection. The server can go back and wait for another connection request on the original socket.

Let us now take a look at the client side. Here, too, a socket must first be created using the socket operation, but explicitly binding the socket to a local address is not necessary, since the operating system can dynamically allocatea port when the connection is set up. The connect operation requires that the caller specifies the transport-level address to which a connection request is to be sent. The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the sendand receive operations. Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close operation. Although there are many exceptions to the rule, the general pattern followed by a client and server for connection-oriented communication using sockets. Details on network programming using sockets and other interfaces in UNIX can be found.

The Message-Passing Interface (MPI)

With the advent of high-performance multicomputer, developers have been looking for message-oriented operations that would allow them to easily write highly efficient applications. This means that the operations should be at a convenient level of abstraction (to ease application development), and that their implementation incurs only minimal overhead. Sockets were deemed insufficient for two reasons. First, they were at the wrong level of abstraction by supporting only simple send and receive operations. Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCP/IP. They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in high-performance server clusters. Those protocols required an interface that could handle more advanced features, such as different forms of buffering and synchronization.

The result was that most interconnection networks and high-performance multicomputer were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication operations. Of course, all libraries were mutually incompatible, so that application developers now had a portability problem. The need to be hardware and platform independent eventually lead to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A (groupID, processID) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address. There may be several, possibly overlapping groups of processes involved in a computation and that are all executing at the same time. Transient asynchronous communication is supported by means of the MPI_bsend operation. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied, the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive operation.

There is also a blocking send operation, called MPI_send, of which the semantics are implementation dependent. The operation MPI_send may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation.

Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI_ssend operation. Finally, the strongest form of synchronous communication is also supported: when a sender calls MPI_sendrecv, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this operation corresponds to a normal RPC.

Both MPI_send and MPI_ssend have variants that avoid copying messages from user buffers to buffers internal to the local MPI runtime system. These variants essentially correspond to a form of asynchronous communication. With MPI_isend, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwriting the message before communication completes, MPI offers operations to check for completion, or even to block if required. As with MPI_send, whether the message has actually been transferred to the receiver or that

it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified.

Likewise, with MPI_issend, a sender also passes only a pointer to the MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it. The operation MPI_recv is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called MPI_irecv, by which a receiver indicates that it is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.

The semantics of MPI communication operations are not always straight- forward, and different operations can sometimes be interchanged without affecting the correctness of a program. The official reason why so many differ- ent forms of communication are supported is that it gives implementers of MPI systems enough possibilities for optimizing performance. Cynics might say the committee could not make up its collective mind, so it threw in every- thing. By now, MPI is in its third version with over 440 operations available. Being designed for high-performance parallel applications, it is perhaps easier to understand its diversity.

Message Oriented Persistent Communication

We now come to an important class of message-oriented middleware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM). Message-queuing systems provide extensive support for persistent asynchronous communication. The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission. An important difference with sockets and MPI is that message-queuing systems are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds.

Message Queuing Model

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent. In practice, most communication servers are directly connected to each other. In other words, a message is generally transferred directly to a destination server. In principle, each application has its own private queue to which other applications can send messages. A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.

An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behavior of the recipient. These semantics permit communication to be loosely coupled in time. There is thus no need for the receiver to be executing when a message is being sent to its queue. Likewise, there is no need for the sender to be executing at the moment its message is picked up by the receiver. The sender and receiver can execute completely independently of each other. In fact, once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This gives us four combinations with respect to the execution mode of the sender and receiver.

Both the sender and receiver execute during the entire transmission of a message. Only the sender is executing,while the receiver is passive, that is, in a state in which message delivery is not possible. Nevertheless, the sender can still send messages. The combination of a passive sender and an executing receiver. In this case, the receiver can read messages that were sent to it, but it is not necessary that their respective senders are executing as well. Finally, we see the situation that the system is storing and possibly transmitting messages even while sender and receiver are passive. One may argue that only if this last configuration is supported, the message-queuing system truly provides persistent messaging.

Messages can, in principle, contain any data. The only important aspect from the perspective of middleware is that messages are properly addressed.In practice, addressing is done by providing a system wide unique name of the destination queue. In some cases, message size may be limited, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications.

The put operation is called by a sender to pass a message to the underlying system that is to be appended to the specified queue. As we explained, this is a non-blocking call. The get operation is a blocking call by which an authorized process can remove the longest pending message in the specified queue. The process is blocked only if the queue is empty. Variations on this call allow searching for a specific message in the queue, for example, using a priority, or a matching pattern. The non-blocking variant is given by the poll operation. If the queue is empty, or if a specific message could not be found, the calling process simply continues.

Finally, most queuing systems also allow a process to install a handler as a callback function, which is automatically invoked whenever a message is putinto the queue. Callbacks can also be used to automatically start a process that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver's sidethat continuously monitors the queue for incoming messages and handles accordingly.

General Architecture of a Message Queuing System

Let us now take a closer look at what a general message-queuing system looks like. First of all, queues are managed by queue managers. A queue manager is either a separate process, or is implemented by means of a library that is linked with an application. Secondly, as a rule of thumb, an application can put messages only into a local queue. Likewise, getting a message is possible by extracting it from a local queue only. As a consequence, if a queue manager QM_A handling the queues for an application a runs as a separate process, both processes QM_A and A will generally be placed on the same machine, or at worst on the same LAN. Also note that if all queue managers are linkedinto their respective applications, we can no longer speak of a persistent asynchronous messaging system.

If applications can put messages only into local queues, then clearly each message will have to carry information concerning its destination. It is the queue manager's task to make sure that a message reaches its destination. This brings us to a number of issues. In the first place, we need to consider how the destination queue is ad- dressed. Obviously, to enhance location transparency, it is preferable that queues have logical, location-independent names. Assuming that a queue man- ager is implemented as a separate process, using logical names implies that each name should be associated with a contact address, such as a pair, and that the name-to-address mapping is readily available to a queue manager. In practice, a contact address carries moreinformation, notably the protocol to be used, such as TCP or UDP.

Multicast Communication

An important topic in communication in distributed systems is the support for sending data to multiple receivers, also known as multicast communication. For many years, this topic has belonged to the domain of network protocols, where numerous proposals for network-level and transport-level solutions have been implemented and evaluated. A major issue in all solutions was setting up the communication paths for information dissemination. In practice, this involved a huge management effort, in many cases requiring human intervention. In addition, as long as there is no convergence of proposals, ISPs have shown to be reluctant to support multicasting. With the advent of peer-to-peer technology, and notably structured overlay management, it became easier to set up communication paths. As peer-to-peer solutions are typically deployed at the application layer, various application-level multicasting techniques have been introduced. In this section, we will take a brief look at these techniques. Multicast communication can also be accomplished in other ways than setting up explicit communication paths.

Application Level Tree based Multicasting

The basic idea in application-level multicasting is that nodes organize into an overlay network, which is then used to disseminate information to its members. An important observation is that network routers are not involved in group membership. As a consequence, the connections between nodes in the overlay network may cross several physical links, and as such, routing messages within the overlay may not be optimal in comparison to what could have been achieved by network-level routing.

Information Dissemination Models

As the name suggests, epidemic algorithms are based on the theory of epidemics, which studies the spreading of infectious diseases. In the case of large-scale distributed systems, instead of spreading diseases, they spread information. Research on epidemics for distributed systems also aims at a completely different goal: whereas health organizations will do their utmost best to prevent infectious diseases from spreading across large groups of people, designers of epidemic algorithms for distributed systems will try to “infect” all nodes with new information as fast as possible.

Using the terminology from epidemics, a node that is part of a distributed system is called infected if it holds data that it is willing to spread to other nodes. A node that has not yet seen this data is called susceptible. Finally, an updated node that is not willing or able to spread its data is said to have been removed. Note that we assume we can distinguish old from new data, for example, because it has been timestamped or versioned. In this light, nodes are also said to spread updates. A popular propagation model is that of anti-entropy. In this model, a node P picks another node Q at random, and subsequently exchanges updates with Q. There are three approaches to exchanging updates:

P only pulls in new updates from Q

P only pushes its own updates to Q

P and Q send updates to each other (i.e., a push-pull approach)

When it comes to rapidly spreading updates, only pushing updates turns out to be a bad choice. Intuitively, this can be understood as follows. First, note that in a pure push-based approach, updates can be propagated only by infected nodes. However, if many nodes are infected, the probability of each one selecting a susceptible node is relatively small.

Consequently, chances are that a particular node remains susceptible for a long period simply because it is not selected by an infected node.

In contrast, the pull-based approach works much better when many nodes are infected. In that case, spreading updates is essentially triggered by susceptible nodes. Chances are big that such a node will contact an infected one to subsequently pull in the updates and become infected as well. If only a single node is infected, updates will rapidly spread across all nodes using either form of anti-entropy, although push-pull remains the best strategy. Define a **round** as spanning a period in which every node will have taken the initiative once to exchange updates with a randomly chosen other node. It can then be shown that the number of rounds to propagate a single update to all nodes takes $(log(N))$, where N is the number of nodes in the system. This indicates indeed that propagating updates is fast, but above all scalable. Specific variant of epidemic protocols is called **rumor spreading**. It works as follows. If node P has just been updated for data item x, it contacts an arbitrary other node Q and tries to push the update to Q. However, it is possible that Q was already updated by another node. In that case, P may lose interest in spreading the update any further, say with probability p_{stop}. In other words, it then becomes removed.

Rumor spreading is gossiping analogous to real life. When Bob has some hot news to spread around, he may phone his friend Alice telling her all about it. Alice, like Bob, will be really excited to spread the rumor to her friends as well. However, she will become disappointed when phoning a friend, say Chuck, only to hear that the news has already reached him. Chances are that she will stop phoning other friends, for what well is it if they already know?

Rumor spreading turns out to be an excellent way of rapidly spread- ing news. However, it cannot guarantee that all nodes will actually be up- dated [Demers et al., 1987]. In fact, when there is a large number of nodes that participate in the epidemics, the fraction s of nodes that will remain ignorant of an update, that is, remain susceptible, satisfies the equation:

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

To get an idea of what this means, take a look at Figure 4.40, which shows s as a function of p_{stop}. Even for high values of p_{stop} we see that the fraction of nodes that remains ignorant is relatively low, and always less that approximately 0.2. For $p_{stop} = 0.20$ it can be shown that $s = 0.0025$. However, in those cases when p_{stop} is relatively high, additional measures will need to be taken to ensure that *all* nodes are updated.

One of the main advantages of epidemic algorithms is their scalability, due to the fact that the number of synchronizations between processes is relatively small compared to other propagation methods. For wide-area systems, Lin and Marzullo [1999] have shown that it makes sense to take the actual network topology into account to achieve better results. In that case, nodes that are connected to only a few other nodes are contacted with a relatively high probability. The underlying assumption is that such nodes form a bridge to other remote parts of the network; therefore, they should be contacted as soon as possible. This approach is referred to as directional gossiping and comes in different variants.

This problem touches upon an important assumption that most epidemic solutions make, namely that a node can randomly select any other node to gossip with. This implies that, in principle, the complete set of nodes should

Removing Data

Epidemic algorithms are extremely good for spreading updates. However, they have a

rather strange side-effect: spreading the *deletion* of a data item is hard. The essence of the problem lies in the fact that deletion of a data item destroys all information on that item. Consequently, when a data item is simply removed from a node, that node will eventually receive old copies of the data item and interpret those as updates on something it did not havebefore.

The trick is to record the deletion of a data item as just another update, and keep a record of that deletion. In this way, old copies will not be interpretedas something new, but merely treated as versions that have been updated bya delete operation. The recording of a deletion is done by spreading **death certificates** of course, the problem with death certificates is that they should eventually be cleaned up, or otherwise each node will gradually build a huge local database of historical information on deleted data items that is otherwise not used. Demers et al. [1987] propose to use what are called dormant death certificates. Each death certificate is timestamped when it is created. If it canbe assumed that updates propagate to all nodes within a known finite time, then death certificates can be removed after this maximum propagation timehas elapsed.

However, to provide hard guarantees that deletions are indeed spread toall nodes, only a very few nodes maintain dormant death certificates that are never thrown away. Assume node P has such a certificate for data item x.If by any chance an obsolete update for x reaches P, P will react by simply spreading the death certificate for x again.

CHAPTER 19

AN OVERVIEW OF NAMING IN DISTRIBUTED SYSTEM

Vineet Saxena, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- tmmit_cool@yahoo.co.in

Names play an important role in all computer systems. They are used to share resources, to uniquely identify entities, to refer to locations, and more. An important issue with naming is that a name can be resolved to the entity it refers to. Name resolution thus allows a process to access the named entity. To resolve names, it is necessary to implement a naming system. The difference between naming in distributed systems and no distributed systems lies in the way naming systems are implemented.

In a distributed system, the implementation of a naming system is itself often distributed across multiple machines. How this distribution is done plays a key role in the efficiency and scalability of the naming system. In this chapter, we concentrate on three different, important ways that names are used in distributed systems.

First, we consider so-called flat-naming systems. In such systems, entities are referred to by an identifier that, in principle, has no meaning at all. In addi- tion, flat names bare no structure, implying that we need special mechanisms to trace the location of such entities. We discuss various approaches, ranging from chains of forwarding links, to distributed hash tables, to hierarchical location services.

In practice, humans prefer to use readable names. Such names are of- ten structured, as is well known from the way Web pages are referred to. Structured names allow for a highly systematic way of finding the server responsible for the named entity, as exemplified by the Domain Name System. We discuss the general principles, as well as scalability issues.

Finally, humans often prefer to describe entities by means of various characteristics, leading to a situation in which we need to resolve a description by means of the attributes assigned to an entity. As we shall see, this typeof name resolution is notoriously difficult, especially in combination with searching. Names, identifiers, and addresses

Let us start by taking a closer look at what a name actually is. A name in a distributed system is a string of bits or characters that is used to refer to an entity. An entity in a distributed system can be practically anything. Typical examples include resources such as hosts, printers, disks, and files. Other well-known examples of entities that are often explicitly named are processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on.

Entities can be operated on. For example, a resource such as a printer offers an interface containing operations for printing a document, requesting the status of a print job, and the like. Furthermore, an entity such as a network connection may provide operations for sending and receiving data, setting quality-of-service parameters, requesting the status, and so forth. To operate on an entity, it is necessary to access it, for which we need an access point. An access point is yet another, but special, kind of entity in a distributed system. The name of an access point is called an address. The address of an access point of an entity is also simply called an address of thatentity.

An entity can offer more than one access point. As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address. Indeed, many people nowadays have several telephone numbers, each number corresponding to a point where they can be reached. In a distributed system, a typical example of an access point is a host running a specific server, with its address formed by the combination of, for example, an IP address and port number (i.e., the server's transport-level address).

An entity may change its access points in the course of time. For example, when a mobile computer moves to another location, it is often assigned a different IP address than the one it had before. Likewise, when a person moves to another city or country, it is often necessary to change telephone numbers as well. In a similar fashion, changing jobs or Internet Service Providers, means changing your e-mail address.

An address is thus just a special kind of name: it refers to an access point of an entity. Because an access point is tightly associated with an entity, it would seem convenient to use the address of an access point as a regular name for the associated entity. Nevertheless, this is hardly ever done as such naming is generally very inflexible and often human unfriendly.

For example, it is not uncommon to regularly reorganize a distributed system so that a specific server is now running on a different host than previously. The old machine on which the server used to be running maybe reassigned to a completely different server. In other words, an entity may easily change an access point, or an access point may be reassigned to a different entity. If an address is used to refer to an entity, we will have an invalid reference the instant the access point changes or is reassigned to another entity. Therefore, it is much better to let a service be known by a separate name independent of the address of the associated server.

Likewise, if an entity offers more than one access point, it is not clear which address to use as a reference. For instance, many organizations distribute their Web service across several servers. If we would use the addresses of those servers as a reference for the Web service, it is not obvious which address should be chosen as the best one. Again, a much better solution is to have a single name for the Web service independent from the addresses of the different Web servers.

These examples illustrate that a name for an entity that is independent from its addresses is often much easier and more flexible to use. Such a name is called location independent. In addition to addresses, there are other types of names that deserve special treatment, such as names that are used to uniquely identify an entity. A true identifier is a name that has the following properties[26].

1. An identifier refers to at most one entity.
2. Each entity is referred to by at most one identifier.
3. An identifier always refers to the same entity (i.e., it is never reused).

By using identifiers, it becomes much easier to unambiguously refer to an entity. For example, assume two processes each refer to an entity by means of an identifier. To check if the processes are referring to the same entity, it is sufficient to test if the two identifiers are equal. Such a test would not be sufficient if the two processes were using regular, no unique, no identifying names. For example, the name "John Smith" cannot be taken as a unique reference to just a single person.

Likewise, if an address can be reassigned to a different entity, we cannot use an address as an identifier. Consider the use of telephone numbers, which are reasonably stable in the sense that a telephone number will often for some time refer to the same person or organization. However, using a telephone number as an identifier will not work, as it can be reassigned in the course of time. Consequently, Bob's new bakery may be receiving phone calls for Alice's old antique store for a long time. In this case, it would have been better to use a true identifier for Alice instead of her phone number.

Addresses and identifiers are two important types of names that are each used for very different purposes. In many computer systems, addresses and identifiers are represented in machine-readable form only, that is, in the formof bit strings. For example, an Ethernet address is essentially a random stringof 48 bits. Likewise, memory addresses are typically represented as 32-bit or 64-bit strings.

Another important type of name is that which is tailored to be used by humans, also referred to as human-friendly names. In contrast to addresses and identifiers, a human-friendly name is generally represented as a character string. These names appear in many different forms. For example, files in UNIX systems have character-string names that can generally be as long as 255 characters, and which are defined entirely by the user. Similarly, DNS names are represented as relatively simple case-insensitive character strings. Having names, identifiers, and addresses brings us to the central theme of this chapter: how do we resolve names and identifiers to addresses? Before we go into various solutions, it is important to realize that there is often a close relationship between name resolution in distributed systems and message routing. In principle, a naming system maintains a name-to- address binding which in its simplest form is just a table of (name, address) pairs. However, in distributed systems that span large networks and for which many resources need to be named, a centralized table is not going to work.

Instead, what often happens is that a name is decomposed into several parts such as ftp.cs.vu.nl. and that name resolution takes place through a recursive lookup of those parts. For example, a client needing to know the address of the FTP server named by ftp.cs.vu.nl. would first resolve nl to findthe server NS(nl) responsible for names that end with nl, after which the restof the name is passed to server NS(nl). This server may then resolve the name vu to the server NS(vu.nl) responsible for names that end with vu.nl. who can further handle the remaining name ftp.cs. Eventually, this leads to routing the name resolution request as:

NS(.) → NS(nl) → NS(vu.nl) → address of ftp.cs.vu.nl

where NS(.) denotes the server that can return the address of NS(nl), also known as the root server. NS(vu.nl) will return the actual address of the FTP server. It is interesting to note that the boundaries between name resolution and message routing are starting to blur.

Flat naming

Above, we explained that identifiers are convenient to uniquely represent entities. In many cases, identifiers are simply random bit strings, which we conveniently refer to as unstructured, or flat names. An important property of such a name is that it does not contain any information whatsoever on howto locate the access point of its associated entity. In the following, we will take a look at how flat names can be resolved, or, equivalently, how we can locate an entity when given only its identifier.

Broadcasting

Consider a distributed system built on a computer network that offers efficient broadcasting facilities. Typically, such facilities are offered by local-area networks in which all machines are connected to a single cable or the logicalequivalent thereof. Also, local-area wireless networks fall into this category.

Locating an entity in such an environment is simple: a message containing the identifier of the entity is broadcast to each machine and each machine is requested to check whether it has that entity. Only the machines that can offer an access point for the entity send a reply message containing the address of that access point.

This principle is used in the Internet **Address Resolution Protocol** (**ARP**) to find the data-link address of a machine when given only an IP address. In essence, a machine broadcasts a packet on the local network asking who the owner of a given IP address is. When the message arrives at a machine, the receiver checks whether it should listen to the requested IP address. If so, it sends a reply packet containing, for example, its Ethernet address. Broadcasting becomes inefficient when the network grows. Not only is network bandwidth wasted by request messages, but, more seriously, too many hosts may be interrupted by requests they cannot answer. One possible solution is to switch to multicasting, by which only a restricted group of hosts receives the request. For example, Ethernet networks support data-link level multicasting directly in hardware.

Multicasting can also be used to locate entities in point-to-point networks. For example, the Internet supports network-level multicasting by allowing hosts to join a specific multicast group. Such groups are identified by a multicast address. When a host sends a message to a multicast address, the network layer provides a best-effort service to deliver that message to all group members.

A multicast address can be used as a general location service for multipleentities. For example, consider an organization where each employee has his or her own mobile computer. When such a computer connects to the locally available network, it is dynamically assigned an IP address. In addition, it joins a specific multicast group. When a process wants to locate computer A, it sends a “where is A?” request to the multicast group. If A is connected, it responds with its current IP address.

Another way to use a multicast address is to associate it with a replicatedentity, and to use multicasting to locate the nearest replica. When sendinga request to the multicast address, each replica responds with its current (normal) IP address. A crude way to select the nearest replica is to choose the one whose reply comes in first, but as it turns out, selecting a nearest replicais generally not that easy.

Forwarding pointers

Another popular approach to locating mobile entities is to make use of for- warding pointers the principle is simple: when an entity moves from A to B, it leaves behind in A a reference to its new location at B. The main advantage of this approach is its simplicity: as soon as an entity hasbeen located, for example by using a traditional naming service, a client can look up the current address by following the chain of forwarding pointers.

There are also drawbacks. First, if no special measures are taken, a chain for a highly mobile entity can become so long that locating that entity is prohibitively expensive. Second, all intermediate locations in a chain will have to maintain their part of the chain of forwarding pointers as long as needed. A third drawback is the vulnerability to broken

links. As soon as any forwarding pointer is lost, the entity can no longer be reached. An important issue is, therefore, to keep chains relatively short, and to ensure that forwarding pointers are robust [27].

Distributed Hash Tables

Let us now take a closer look at how to resolve an identifier to the address of the associated entity. We have already mentioned distributed hash tables a number of times, but have deferred discussion on how they actually work. In this section we correct this situation by first considering the Chord system asan easy-to-explain DHT-based system as display.

In large distributed systems the collection of participating nodes can be expected to change all the time. Not only will nodes join and leave voluntarily, we also need to consider the case of nodes failing and thus effectively leaving the system, to later recover again at which point they rejoin. Joining a DHT-based system such as Chord is relatively simple. Suppose node p wants to join. It simply contacts an arbitrary node in the existing system and requests a lookup for succ (p + 1). Once this node has been identified, p can insert itself into the ring. Likewise, leaving can be just as simple. Note that nodes also keep track of their predecessor.

Obviously, the complexity comes from keeping the finger tables up-to- date. Most important is that for every node q, $FT_q[1]$ is correct as this entry refers to the next node in the ring, that is, the successor of q + 1. In order to achieve this goal, each node q regularly runs a simple procedure that contacts succ(q + 1) and requests to return pred(succ(q + 1)). If q = pred(succ(q + 1)) then q knows its information is consistent with that of its successor. Otherwise, if q's successor has updated its predecessor, then apparently a new node p had entered the system, with $q < p$ succ(q + 1), so that q will adjust $FT_q[1]$ to p. At that point, it will also check whether p has recorded q as its predecessor. If not, another adjustment of $FT_q[1]$ is needed. In a similar way, to update a finger table, node q simply needs to find the successor for $k = q + 2^{i-1}$ for each entry i. Again, this can be done by issuing a request to resolve succ(k). In Chord, such requests are issued regularly by means of a background process.

Likewise, each node q will regularly check whether its predecessor is alive.If the predecessor has failed, the only thing that q can do is record the fact by setting pred(q) to "unknown." On the other hand, when node q is updating its link to the next known node in the ring, and finds that the predecessor of succ(q + 1) has been set to "unknown," it will simply notify succ(q + 1) that it suspects it to be the predecessor. By and large, these simple procedures ensure that a Chord system is generally consistent, only perhaps with exception of a few nodes.

Structured Naming

Flat names are good for machines, but are generally not very convenientfor humans to use. As an alternative, naming systems generally support structured names that are composed from simple, human-readable names. Not only file naming, but also host naming on the Internet follows thisapproach. In this section, we concentrate on structured names and the way that these names are resolved to addresses.

Name Spaces

Names are commonly organized into what is called a name space. Name spaces for structured names can be represented as a labeled, directed graph with two types of nodes. A leaf node

represents a named entity and has the property that it has no outgoing edges. A leaf node generally stores information on the entity it is representing for example, its address so that a client can access it. Alternatively, it can store the state of that entity, such as in the case of file systems in which a leaf node actually contains the complete file it is representing. We return to the contents of nodes below.

In contrast to a leaf node, a directory node has a number of outgoing edges, each labeled with a name. Each node in a naming graph is considered as yet another entity in a distributed system, and, in particular, has an associated identifier. A directory node stores a table in which an outgoing edge is represented as a pair (node identifier, edge label). Such a table is called a directory table.

The naming graph has one node, namely n0, which has only outgoing and no incoming edges. Such a node is called the root (node) of the naming graph. Although it is possible for a naming graph to have several root nodes, for simplicity, many naming systems have only one. Each path in a naming graph can be referred to by the sequence of labels corresponding to the edges in that path, such as N:[$label_1$, $label_2$, ..., $label_n$], where N refers to the first node in the path. Such a sequence is called a path name. If the first node in a path name is the root of the naming graph, it is called an absolute path name. Otherwise, it is called a relative path name.

It is important to realize that names are always organized in a name space. As a consequence, a name is always defined relative only to a directory node. In this sense, the term "absolute name" is somewhat misleading. Likewise, the difference between global and local names can often be confusing. A global name is a name that denotes the same entity, no matter where that name is used in a system. In other words, a global name is always interpreted with respect to the same directory node. In contrast, a local name is a name whose interpretation depends on where that name is being used. Put differently, a local name is essentially a relative name whose directory in which it is contained is (implicitly) known.

This description of a naming graph comes close to what is implemented in many file systems. However, instead of writing the sequence of edge labels to represent a path name, path names in file systems are generally represented as a single string in which the labels are separated by a special separator character, such as a slash ("/"). This character is also used to indicate whether a path name is absolute. Insteadof using n0: [home, steen, mbox], that is, the actual path name, it is common practice to use its string representation /home/steen/mbox. Note also that when there are several paths that lead to the same node, that node can be represented by different path names. For example, node n5 in Figure 5.11 can be referred to by /home/steen/keys as well as /keys. The string representation of path names can be equally well applied to naming graphs other than those used for only file systems. In Plan 9 all resources, such as processes, hosts, I/O devices, and network interfaces, are named in the same fashion as traditional files. This approach is analogous to implementing a single naming graph for all resources in a distributed system.

There are many different ways to organize a name space. As we mentioned, most name spaces have only a single root node. In many cases, a name space is also strictly hierarchical in the sense that the naming graph is organized as a tree. This means that each node except the root has exactly one incoming edge; the root has no incoming edges. As a consequence, each node also has exactly one associated (absolute) path name. The naming graph is an example of directed acyclicgraph. In such an organization, a node can have more than one incoming edge, but the graph is not permitted to have a cycle. There are also name spaces that do not have this restriction.

Name Resolution

Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names. More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name. The process of looking up a name is called name resolution.

To explain how name resolution works, let us consider a path name such as N:[$label_1$, $label_2$, ..., $label_n$]. Resolution of this name starts at node N of the naming graph, where the name $label_1$ is looked up in the directory table, and which returns the identifier of the node to which $label_1$ refers. Resolution then continues at the identified node by looking up the name $label_2$ in its directory table, and so on. Assuming that the named path actually exists, resolution stops at the last node referred to by $label_n$, by returning that node's content.

Closure Mechanism

Name resolution can take place only if we know how and where to start. In our example, the starting node was given, and we assumed we had access to its directory table. Knowing how and where to start name resolution is generally referred to as a closure mechanism. Essentially, a closure mechanism deals with selecting the initial node in a name space from which name resolution is to start. What makes closure mechanisms sometimes hard to understand is that they are necessarily partly implicit and may be very different when comparing them to each other.

As another example, consider the use of global and local names in distributed systems. A typical example of a local name is an environment variable. For example, in UNIX systems, the variable named HOME is used to refer to the home directory of a user. Each user has its own copy of this variable, which is initialized to the global, system wide name corresponding to the user's home directory. The closure mechanism associated with environmentvariables ensures that the name of the variable is properly resolved by looking it up in a user-specific table[28].

Linking and mounting

Strongly related to name resolution is the use of aliases. An alias is another name for the same entity. An environment variable is an example of an alias.In terms of naming graphs, there are basically two different ways to implement an alias. The first approach is to simply allow multiple absolute paths namesto refer to the same node in a naming graph. This approach is illustrated in Figure 52, in which node n5 can be referred to by two different path names. In UNIX terminology, both path names /keys and /home/steen/keysin Figure 52 are called hard links to node n5. The second approach is to represent an entity by a leaf node, say N, but instead of storing the address or state of that entity, the node stores an absolutepath name. When first resolving an absolute path name that leads to N, name resolution will return the path name stored in N, at which point it can continue with resolving that new path name. This principle corresponds to the use of symbolic links in Unix file systems. In this example, the path name /home/steen/keys, which refers to a node containing the absolute path name /keys, is a symbolic link to node n5.

Name resolution as described so far takes place completely within a single name space. However, name resolution can also be used to merge different name spaces in a transparent way. Let us first consider a mounted file system. In terms of our naming model, a mounted file system corresponds to letting a directory node store the identifier of a directory node from a different name space, which we refer to as a foreign name space. The directory node storing

the node identifier is called a mount point. Accordingly, the directory node in the foreign name space is called a mounting point. Normally, the mounting point is the root of a name space. During name resolution, the mounting point is looked up and resolution proceeds by accessing its directory table.

The principle of mounting can be generalized to other name spaces as well. In particular, what is needed is a directory node that acts as a mount point and stores all the necessary information for identifying and accessing the mounting point in the foreign name space. This approach is followed in many distributed file systems. Consider a collection of name spaces that is distributed across different machines. In particular, each name space is implemented by a different server, each possibly running on a separate machine. Consequently, if we want to mount a foreign name space NS_2 into a name space NS_1, it may be necessary to communicate over a network with the server of NS_2, as that server may be running on a different machine than the server for NS_1. To mount a foreign name space in a distributed system requires at least the following information:

1. The name of an access protocol.
2. The name of the server.
3. The name of the mounting point in the foreign name space.

Note that each of these names needs to be resolved. The name of an access protocol needs to be resolved to the implementation of a protocol by which communication with the server of the foreign name space can take place. The name of the server needs to be resolved to an address where that server canbe reached. As the last part in name resolution, the name of the mountingpoint needs to be resolved to a node identifier in the foreign name space.

In no distributed systems, none of the three points may actually be needed. For example, in UNIX there is no access protocol and no server. Also, the name of the mounting point is not necessary, as it is simply the root directory of the foreign name space.

The name of the mounting point is to be resolved by the server of theforeign name space. However, we also need name spaces and implementations for the access protocol and the server name. One possibility is to representthe three names listed above as a URL. To make matters concrete, consider a situation in which a user with a laptop computer wants to access files that are stored on a remote file server.The client machine and the file server are both configured with the Network File System (NFS).

The name NFS is a well-known name in the sense that worldwide agreement exists on how to interpret that name. Given that we are dealing with a URL, the name nfs will be resolved to an implementation of the NFS protocol. The server name is resolved to its address using DNS, which is discussed in a later section. As we said, /home/steen is resolved by the server of the foreign name space.

CHAPTER 20

THE IMPLEMENTATION OF A NAME SPACE

Amit Kumar Bishnoi, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- amit.vishnoi08@gmail.com

A name space forms the heart of a naming service, that is, a service that allows users and processes to add, remove, and look up names. A naming service is implemented by name servers. If a distributed system is restricted to a local-area network, it is often feasible to implement a naming service by means of only a single name server. However, in large-scale distributed systems with many entities, possibly spread across a large geographical area,it is necessary to distribute the implementation of a name space over multiple name servers.

Name Space Distribution

Name spaces for a large-scale, possibly worldwide distributed system, are usually organized hierarchically. As before, assume such a name space has only a single root node. To effectively implement such a name space, itis convenient to partition it into logical layers.

The global layer is formed by highest-level nodes, that is, the root node and other directory nodes logically close to the root, namely its children. Nodes in the global layer are often characterized by their stability, in the sense that directory tables are rarely changed. Such nodes may represent organizations, or groups of organizations, for which names are stored in the name space. The administrational layer is formed by directory nodes that together are managed within a single organization. A characteristic feature of the directory nodes in the administrational layer is that they represent groups of entities that belong to the same organization or administrational unit. For example, there may be a directory node for each department in an organization, or a directory node from which all hosts can be found. Another directory node may be used as the starting point for naming all users, and so forth. The nodes in the administrational layer are relatively stable, although changes generally occur more frequently than to nodes in the global layer [29].

Finally, the managerial layer consists of nodes that may typically change regularly. For example, nodes representing hosts in the local network belongto this layer. For the same reason, the layer includes nodes representing shared files such as those for libraries or binaries. Another important class of nodes includes those that represent user-defined directories and files. In contrast to the global and administrational layer, the nodes in the managerial layer are maintained not only by system administrators, but also by individual endusers of a distributed system.

To make matters more concrete, an example of the partitioning of part of the DNS name space, including the names of files within an organization that can be accessed through the Internet, for example, Webpages and transferable files. The name space is divided into no overlapping parts, called **zones** in DNS. A zone is a part of the name space that is implemented by a separate name server. If we take a look at availability and performance, name servers in each layer have to meet different requirements. High availability is especially critical for name servers in the global layer.

If a name server fails, a large part of the name space will be unreachable because name resolution cannotproceed beyond the failing server. Performance is somewhat subtle. Due to the low rate of change of nodes in the global layer, the results of lookup operations generally remain valid for a long time. Consequently, those results can be effectively cached by the clients. The next time the same lookup operation is performed,the results can be retrieved from the client's cache instead of letting the nameserver return the results. As a result, name servers in the global layer do not have to respond quickly to a single lookup request. On the other hand, throughput may be important, especially in large-scale systems with millions of users.

The availability and performance requirements for name servers in the global layer can be met by replicating servers, in combination with client-side caching. Updates in this layer generally do not have to come into effect immediately, making it much easier to keep replicas consistent. Availability for a name server in the administrational layer is primarily important for clients in the same organization as the name server. If the name server fails, many resources within the organization become unreachable because they cannot be looked up. On the other hand, it may be less important that resources in an organization are temporarily unreachable for users outsidethat organization.

With respect to performance, name servers in the administrational layer have similar characteristics as those in the global layer. Because changes to nodes do not occur all that often, caching lookup results can be highly effective, making performance less critical. However, in contrast to the global layer, the administrational layer should take care that lookup results are returned within a few milliseconds, either directly from the server or from the client's local cache. Likewise, updates should generally be processed quicker than those of the global layer. For example, it is unacceptable that an account for anew user takes hours to become effective.

These requirements can often be met by using relatively powerful machines to run name servers. In addition, client-side caching should be applied, combined with replication for increased overall availability. Availability requirements for name servers at the managerial level are generally less demanding. In particular, it often suffices to use a single machine to run name servers at the risk of temporary unavailability. However, performance is crucial: operations must take place immediately. Because updates occur regularly, client-side caching is often less effective.

Table 1: Represented that A comparison between name servers for implementing nodes from a large-scale name space partitioned into a global layer, an administrational layer, and a managerial layer.

Issue	Global	Administrational	Managerial
Geographical scale	Worldwide	Organization	Department
Number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Client-side caching	Yes	Yes	Sometimes

A comparison between name servers at different layers is shown in Table 1. In distributed systems, name servers in the global and administrational layer are the most difficult to implement. Difficulties are caused by replication and caching, which are needed for availability and performance, but which also introduce consistency problems. Some of the problems are aggravated by the fact that caches and replicas are spread across a wide-area network, which may introduce long communication delays during lookups.

The DNS Name Space

The DNS name space is hierarchically organized as a rooted tree. A label is a case-insensitive string made up of alphanumeric characters. A label has a maximum length of 63 characters; the length of a complete path name is restricted to 255 characters. The string representation of a path name consists of listing its labels, starting with the rightmost one, and separating the labels by a dot (“.”). The root is represented by a dot. So, for example, the path name root: [nl, vu, cs, flits], is represented by the string “flits.cs.vu.nl.”, which includes the rightmost dot to indicate the root node. We generally omit this dot for readability because each node in the DNS name space has exactly one incoming edge (with the exception of the root node, which has no incoming edges), the label attached to a node’s incoming edge is also used as the name for that node. A subtree is called a domain a path name to its root node is called a domain name. Note that, just like a path name, a domain name can be either absolute or relative. The contents of a node is formed by a collection of resource records. There are different types of resource records. The major ones are shown in Table 2.

Table 2: Represented that the most important types of resource records forming the con-tents of nodes in the DNS name space.

Type	Refers to	Description
SOA	Zone	Holds info on the represented zone
A	Host	IP addr. of host this node represents
MX	Domain	Mail server to handle mail for this node
SRV	Domain	Server handling a specific service
NS	Zone	Name server for the represented zone
CNAME	Node	Symbolic link
PTR	Host	Canonical name of a host
HINFO	Host	Info on this host
TXT	Any kind	Any info considered useful

A node in the DNS name space will often represent several entities at the same time. For example, a domain name such as vu.nl is used to represent a domain and a zone. In this case, the domain is implemented by means of several zones. An SOA (start of authority)

resource record contains information such as an e-mail address of the system administrator responsible for the represented zone, the name of the host where data on the zone can be fetched, and so on.

An A (address) record, represents a particular host in the Internet. The Arecord contains an IP address for that host to allow communication. If a hosthas several IP addresses, as is the case with multi-homed machines, the node will contain an A record for each address. Another type of record is the MX (mail exchange) record, which is likea symbolic link to a node representing a mail server. For example, the noderepresenting the domain cs.vu.nl has an MX record containing the name zephyr.cs.vu.nl which refers to a mail server. That server will handle all incoming mail addressed to users in the cs.vu.nl domain. There may be several MXrecords stored in a node [30].

Related to MX records are SRV records, which contain the name of a server for a specific service. The service itself is identified by means of a name along with the name of a protocol. For example, the Web server in the cs.vu.nl domain could be named by means of an SRV record such as _http_tcp.cs.vu.nl. This record would then refer to the actual name of the server (which is soling.cs.vu.nl). An important advantage of SRV records is that clients need no longer know the DNS name of the host providing a specific service. Instead, only service names need to be standardized, after which the providing host can be looked up.

Nodes that represent a zone, contain one or more NS (name server) records. Like MX records, an NS record contains the name of a name server that implements the zone represented by the node. In principle, each node in the name space can store an NS record referring to the name server that implements it. However, as we discuss below, the implementation of the DNS name space is such that only nodes representing zones need to store NSrecords.

DNS distinguishes aliases from what are called **canonical names**. Each host is assumed to have a canonical, or primary name. An alias is implemented by means of node storing a CNAME record containing the canonical name of a host. The name of the node storing such a record is thus the same as a symbolic link. DNS maintains an inverse mapping of IP addresses to host names by means of PTR (pointer) records. To accommodate the lookups of host names when given only an IP address, DNS maintains a domain named in-addr.arpa, which contains nodes that represent Internet hosts and which are named bythe IP address of the represented host. For example, host www.cs.vu.nl hasIP address 130.37.20.20. DNS creates a node named 20.20.37.130.in-addr.arpa, which is used to store the canonical name of that host which happens to be soling.cs.vu.nl in a PTR record. Finally, an HINFO (host info) record is used to store additional information on a host such as its machine type and operating system. In a similar fashion, TXT records are used for any other kind of data that a user finds useful to store about the entity represented by the node.

DNS Implementation

In essence, the DNS name space can be divided into a global layer and an administrational layer. The managerial layer, which is generally formed by local file systems, is formally not part of DNS and is therefore also not managed by it. Each zone is implemented by a name server, which is virtually always replicated for availability. Updates for a zone are normally handled by the primary name server. Updates take place by modifying the DNS database local to the primary server. Secondary name servers do not access the database directly, but, instead, request the primary server to transfer its content. The latter is called a **zone transfer** in DNS terminology.

A DNS database is implemented as a (small) collection of files, of which the most important one contains all the resource records for *all* the nodes in a particular zone. This approach allows nodes to be simply identified by means of their domain name, by which the notion of a node identifier reduces to an (implicit) index into a file. Web server, as well as the department's FTP server are implemented by a single machine, called soling.cs.vu.nl. By executing both servers on the same machine (and essentially using that machine only for Internet services and not anything else), system management becomes easier. For example, both servers will have the same view of the file system, and for efficiency, part of the file system may be implemented on soling.cs.vu.nl. This approach is often applied in the case of WWW and FTP services.

The Network File System

As another, and very different example, consider naming in NFS. The funda- mental idea underlying the NFS naming model is to provide clients complete transparent access to a remote file system as maintained by a server. This transparency is achieved by letting a client be able to mount a remote file system into its own local file system.

Instead of mounting an entire file system, NFS allows clients to mount only part of a file system, as also shown in Figure 7. A server is said to **export** a directory when it makes that directory and its entries available to clients. An exported directory can be mounted into a client's local name space. This design approach has a serious implication: in principle, users do not share name spaces. The file named /remote/vu/mboxat client A is named /work/me/mbox at client B. A file's name therefore depends on how clients organize their own local name space, and where exported directories are mounted. The drawback of this approach in a distributed file system is that sharing files becomes much harder. For example, Alice cannot tell Bob about a file using the name she assigned to that file, forthat name may have a completely different meaning in Bob's name space of files.

There are several ways to solve this problem, but the most common oneis to provide each client with a name space that is partly standardized. For example, each client may be using the local directory /usr/bin to mount a file system containing a standard collection of programs that are available to everyone. Likewise, the directory /local may be used as a standard to mounta local file system that is located on the client' host.

An NFS server can itself mount directories that are exported by other servers. However, it is not allowed to export those directories to its own clients. Instead, a client will have to explicitly mount such a directory from the server that maintains it, as shown in Figure 8. This restriction comes partly from simplicity. If a server could export a directory that it mounted from another server, it would have to return special file handles that includean identifier for a server. NFS does not support such file handles.

FS_A from which it exports the directory /packages. This directory contains a subdirectory /draw that acts as a mount point for a file system FS_B that is exported by server B and mounted by A. Let A also export /packages/draw to its own clients, and assume that a client has mounted /packages into its local directory /bin. If name resolution is iterative, then to resolve the name /bin/draw/install, the client contacts server A when it has locally resolved /bin and requests A to return a file handle for directory /draw. In that case, server A should return a file handle that includes an identifier for server B, for only B can resolve the rest of the path name, in this case /install. As we have said, this kind of name resolution is not supported by NFS.

Name resolution in earlier versions of NFS is strictly iterative in the sense that only a single file name at a time can be looked up. In other words, resolving a name such as /bin/draw/install

requires three separate calls to the NFS server. Moreover, the client is fully responsible for implementing the resolution of a path name. NFSv4 also supports recursive name lookups. In this case, a client can pass a complete path name to a server and request that server to resolve it.

There is another peculiarity with NFS name lookups that has been solved with the most recent version (NFSv4). Consider a file server hosting several file systems. With the strict iterative name resolution, whenever a lookup is done for a directory on which another file system was mounted, the lookup would return the file handle of the directory. Subsequently reading that directory would return its original content, not that of the root directory of the mounted file system.

To explain, assume that in our previous example that both file systems FSA and FSB are hosted by a single server. If the client has mounted /packages into its local directory /bin, then looking up the file name draw at the server would return the file handle for draw. A subsequent call to the server for listing the directory entries of draw by means of readdir would then return the list of directory entries that were originally stored in FSA in subdirectory/packages/draw. Only if the client had also mounted file system FSB, would it be possible to properly resolve the path name draw/install relative to /bin.

NFSv4 solves this problem by allowing lookups to cross mount points at a server. In particular, lookup returns the file handle of the mounted directory instead of that of the original directory. The client can detect that the lookup has crossed a mount point by inspecting the file system identifier of the looked up file. If required, the client can locally mount that file system as well. A file handle is a reference to a file within a file system. It is independent of the name of the file it refers to. A file handle is created by the server that is hosting the file system and is unique with respect to all file systems exported by the server. It is created when the file is created. The client is kept ignorant of the actual content of a file handle; it is completely opaque.

Ideally, a file handle is implemented as a true identifier for a file relative to a file system. For one thing, this means that as long as the file exists, it should have one and the same file handle. This persistence requirement allows a client to store a file handle locally once the associated file has been looked up by means of its name. One benefit is performance: as most file operations require a file handle instead of a name, the client can avoid having to look up a name repeatedly before every file operation. Another benefit of this approach is that the client can now access the file regardless which (current) name it has because a file handle can be locally stored by a client, it is also important that a server does not reuse a file handle after deleting a file. Otherwise, a client may mistakenly access the wrong file when it uses its locally stored file handle.

Note that the combination of iterative name lookups and not letting a lookup operation allow crossing a mount point introduces a problem with getting an initial file handle. In order to access files in a remote file system, a client will need to provide the server with a file handle of the directory where the lookup should take place, along with the name of the file or directory that is to be resolved. NFSv3 solves this problem through a separate mount protocol, by which a client actually mounts a remote file system. After mounting, the client is passed back the **root file handle** of the mounted file system, which it can subsequently use as a starting point for looking up names.

In NFSv4, this problem is solved by providing a separate operation put roofs that tells the server to solve all file names relative to the root file handle of the file system it manages.

The root file handle can be used to look up any other file handle in the server's file system. This approach has the additional benefit that there is no need for a separate mount protocol. Instead, mounting can be integrated into the regular protocol for looking up files. A client can simply mount a remote file system by requesting the server to resolve names relative to the file system's root file handle using putrootfh.

Attribute Based Naming

Flat and structured names generally provide a unique and location-independent way of referring to entities. Moreover, structured names have been partly designed to provide a human-friendly way to name entities so that they can be conveniently accessed. In most cases, it is assumed that the name refers to only a single entity. However, location independence and human friendliness are not the only criterion for naming entities. In particular, as more information is being made available it becomes important to effectively search for entities. This approach requires that a user can provide merely a description of what he is looking for.

There are many ways in which descriptions can be provided, but a popular one in distributed systems is to describe an entity in terms of (attribute, value) pairs, generally referred to as attribute-based naming. In this approach, an entity is assumed to have an associated collection of attributes. Each attribute says something about that entity. By specifying which values a specific attribute should have, a user essentially constrains the set of entities that he is interested in. It is up to the naming system to return one or more entities that meet the user's description. In this section we take a closer look at attribute-based naming systems.

Directory services

Attribute-based naming systems are also known as **directory services**, whereas systems that support structured naming are generally called **naming systems**. With directory services, entities have a set of associated attributes that can be used for searching. In some cases, the choice of attributes can be relatively simple. For example, in an e-mail system, messages can be tagged with attributes for the sender, recipient, subject, and so on. However, even in the case of e-mail, matters become difficult when other types of descriptors are needed, as is illustrated by the difficulty of developing filters that will allow only certain messages (based on their descriptors) to be passed through. What it all boils down to is that designing an appropriate set of attributes is not trivial. In most cases, attribute design has to be done manually. Even if there is consensus on the set of attributes to use, practice shows that setting the values consistently by a diverse group of people is a problem by itself, as many will have experienced when accessing music and video databases on the Internet.

To alleviate some of these problems, research has been conducted on unifying the ways that resources can be described. In the context of distributed systems, one particularly relevant development is the **resource description framework** (**RDF**). Fundamental to the RDF model is that resources are described as triplets consisting of a subject, a predicate, and an object. For example, (Person, name, Alice) describes a resource named Person whose name is Alice. In RDF, each subject, predicate, or object can be a resource itself. This means that Alice may be implemented as a reference to a file that can be subsequently retrieved. In the case of a predicate, such a resource could contain a textual description of that predicate. Resources associated with subjects and objects can be anything. References in RDF are essentially URLs.

If resource descriptions are stored, it becomes possible to query that storage in a way that is common for many attribute-based naming systems. For example, an application could

ask for the information associated with a person named Alice. Such a query would return a reference to the person resource associated with Alice. This resource can then subsequently be fetched by the application.

In this example, the resource descriptions are stored at a central location. There is no reason why the resources should reside at the same location as well. However, not having the descriptions in the same place may incur a serious performance problem. Unlike structured naming systems, looking up values in an attribute-based naming system essentially requires an exhaustive search through all descriptors. Various techniques can be applied to avoid such exhaustive searches, one obvious being indexing. When considering performance, an exhaustive search may be less of problem within a single, non-distributed data store, but simply sending a search query to hundreds of servers that jointly implement a distributed data store is generally not such a good idea. In the following, we will take a look at different approaches to solving this problem in distributed systems.

Hierarchical implementations: LDAP

A common approach to tackling distributed directory services is to combine structured naming with attribute-based naming. This approach has been widely adopted, for example, in Microsoft's Active Directory service and other systems. Many of these systems use, or rely on the lightweight directory access protocol commonly referred simply as LDAP. The LDAP directory service has been derived from OSI's X.500 directory service. As with many OSI services, the quality of their associated implementations hindered widespread use, and simplifications were needed to make it useful. Detailed information on LDAP.

Conceptually, an LDAP directory service consists of a number of records, usually referred to as directory entries. A directory entry is comparable to a resource record in DNS. Each record is made up of a collection of (at- tribute, value) pairs, where each attribute has an associated type. A distinction is made between single-valued attributes and multiple-valued attributes. The latter typically represent arrays and lists. As an example, a simple directory entry identifying the network addresses of some general servers from Figure 5.23 is shown in Table 1.

Table 1: Represented that the simple example of an LDAP directory entry using LDAP naming conventions.

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	VU University
Organizational Unit	OU	Computer Science
Common Name	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20
WWW_Server	—	130.37.20.20

In our example, we have used a naming convention described in the LDAP standards, which applies to the first five attributes. The attributes Organization and Organization Unit describe, respectively, the organization and the department associated with the data that are stored in the record. Likewise, the attributes Locality and Country provide additional information on where the entry is stored. The Common Name attribute is often used as an (ambiguous) name to identify an entry within a limited part of the directory. For example, the name "Main server" may be enough to find our example entry given the specific values for the other four attributes Country, Locality, Organization, and Organizational Unit. In our example, only attribute Mail_Servers has multiple values associated with it. All other attributes have only a single value.

The collection of all directory entries in an LDAP directory service is called a directory information base (DIB). An important aspect of a DIB is that each record is uniquely named so that it can be looked up. Such a globally unique name appears as a sequence of naming attributes in each record. Each naming attribute is called a relative distinguished name, or RDN for short. In our example in Figure 5.28 the first five attributes are all naming attributes. Using the conventional abbreviations for representing naming attributes in LDAP, as shown in Figure 5.28 the attributes Country, Organization, and Organizational Unit could be used to form the globally unique name /C = NL/O = VU University/OU = Computer Science. Analogous to the DNS name nl.vu.cs.

As in DNS, the use of globally unique names by listing RDNs in sequence, leads to a hierarchy of the collection of directory entries, which is referred to as a **directory information tree** (**DIT**). A DIT essentially forms the naming graph of an LDAP directory service in which each node represents a directory entry. In addition, a node may also act as a directory in the traditional sense, in that there may be several children for which the node acts as parent. To explain, consider the naming graph as partly shown in Figure 56. (Recall that labels are associated with edges.)

Node N corresponds to the directory entry at the same time, this node acts as a parent to a number of other directory entries that have an additional naming attribute Hostname that is used as an RDN. For example, such entries may be used to represent hosts. A node in an LDAP naming graph can thus simultaneously represent a directory in the traditional sense as we discussed previously, as well as an LDAP record. This distinction is supported by two different lookup operations. The read operation is used to read a single record given its path name in the DIT. In contrast, the list operation is used to list the names of all outgoing edges of a given node in the DIT.

Implementing an LDAP directory service proceeds in much the same way as implementing a naming service such as DNS, except that LDAP supports more lookup operations as we will discuss shortly. When dealing with a large- scale directory, the DIT is usually partitioned and distributed across several servers, known as directory service agents (DSA). Each part of a partitioned DIT thus corresponds to a zone in DNS. Likewise, each DSA behaves very much the same as a normal name server, except that it implements a number of typical directory services, such as advanced search operations.

Clients are represented by what are called directory user agents, or simply **DUA**. A DUA is similar to a name resolver in structured-naming services. A DUA exchanges information with a DSA according to a standardized access protocol. What makes an LDAP implementation different from a DNS implementation are the facilities for searching through a DIB. In particular, facilities are provided to search for a directory entry given a set of

criteria that attributes of the searched entries should meet. For example, suppose that we want a list of all main servers at VU University. Using the notation defined in Howes [1997], such a list can be returned using a search operation like

search(''(C=NL)(O=VU University)(OU=*)(CN=Main server)'')

In this example, we have specified that the place to look for main servers is the organization named VU_University in country NL, but that we are not interested in a particular organizational unit. However, each returned result should have the CN attribute equal to Main_server.

As we already mentioned, searching in a directory service is generally an expensive operation. For example, to find all main servers at VU University requires searching all entries at each department and combining the results in a single answer. In other words, we will generally need to access several leaf nodes of a DIT in order to get an answer. In practice, this also means that several DSAs need to be accessed. In contrast, naming services can often be implemented in such a way that a lookup operation requires accessing only a single leaf node.

This whole setup of LDAP can be taken one step further by allowing several trees to co-exist, while also being linked to each other. This approach is followed in Microsoft's Active Directory leading to a forest of LDAP domains [Allen and Lowe-Norris, 2003]. Obviously, searching in such an organization can be overwhelmingly complex. To circumvent some of the scalability problems, Active Directory usually assumes there is a global index server (called a global catalog) that can be searched first. The index will indicate which LDAP domains need to be searched further.

Although LDAP by itself already exploits hierarchy for scalability, it is common to combine LDAP with DNS. For example, every tree in LDAP needs to be accessible at the root (known in Active Directory as a domain controller). The root is often known under a DNS name, which, in turn, can be found through an appropriate SRV record as we explained above.

Decentralized Implementations

Notably with the advent of peer-to-peer systems, researchers have also been looking for solutions for decentralized attribute-based naming systems. In particular, peer-to-peer systems are often used to store files. Initially, files could not be searched—they could only be looked up by their key. However, having the possibility to search for a file based on descriptors can be extremely convenient, where each descriptor is nothing but an (attribute, value) pair. Obviously, querying every node in a peer-to-peer system to see if it contains a file matching one or more of such pairs is infeasible. What we need is a mapping of (attribute, value) pairs to index servers, which, in turn, point to files matching those pairs.

CHAPTER 21

AN INTRODUCTION OF COORDINATION

Shambhu Bhardwaj, Associate Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- shambhu.bharadwaj@gmail.com

In the previous chapters, we have looked at processes and communication between processes. While communication is important, it is not the entire story. Closely related is how processes cooperate and synchronize with one another. Cooperation is partly supported by means of naming, which allows processes to at least share resources, or entities in general. In this chapter, we mainly concentrate on how processes can synchronize and coordinate their actions. For example, it is important that multiple processes do not simultaneously access a shared resource, such as a file, but instead cooperate in granting each other temporary exclusive access. Another example is that multiple processes may sometimes need to agree on the ordering of events, such as whether message m_1 from process P was sent before or after message m_2 from process Q.

Synchronization and coordination are two closely related phenomena. In process synchronization we make sure that one process waits for another to complete its operation. When dealing with data synchronization, the problem is to ensure that two sets of data are the same. When it comes to coordination, the goal is to manage the interactions and dependencies between activities in a distributed system. From this perspective, one could state that coordination encapsulates synchronization [31].

As it turns out, coordination in distributed systems is often much more difficult compared to that in uniprocessor or multiprocessor systems. The problems and solutions that are discussed in this chapter are, by their nature, rather general, and occur in many different situations in distributed systems. We start with a discussion of the issue of synchronization based on actual time, followed by synchronization in which only relative ordering matters rather than ordering in absolute time. In many cases, it is important that a group of processes can appoint one process as a coordinator, which can be done by means of election algorithms. We discuss various election algorithms in a separate section.

Before that, we look into a number of algorithms for coordinating mutual exclusion to a shared resource. As a special class of coordination problems, we also dive into location systems by which we place a process in a multidimensional plane. Such placements come in handy when dealing with very large distributed systems.

We already came across publish-subscribe systems, but have not yet dis-cussed in any detail how we actually match subscriptions to publications. There are many ways to do this and we look at centralized as well as decentralized implementations. Finally, we consider three different gossip-based coordination problems: aggregation, peer sampling, and overlay construction.

Clock synchronization

In a centralized system, time is unambiguous. When a process wants to know the time, it simply makes a call to the operating system. If process A asks for the time, and then a little

later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower. In a distributed system, achieving agreement on time is not trivial. Just think, for a moment, about the implications of the lack of global timeon the UNIX make program, as a simple example. Normally, in Unix large programs are split up into multiple source files, so that a change to one source file requires only one file to be recompiled, not all the files. If a program consists of 100 files, not having to recompile everything because one file hasbeen changed greatly increases the speed at which programmers can work.

The way make normally works is simple. When the programmer has finished changing all the source files, he runs make, which examines the times at which all the source and object files were last modified. If the source file input.c has time 2151 and the corresponding object file input.o has time2150, make knows that input.c has been changed since input.o was created, and thus input.c must be recompiled. On the other hand, if output.c hastime 2144 and output.o has time 2145, no compilation is needed. Thus make goes through all the source files to find out which ones need to be recompiledand calls the compiler to recompile them.

Now imagine what could happen in a distributed system in which there was no global agreement on time. Suppose that output has time 2144 as above, and shortly thereafter output is modified but is assigned time 2143 because the clock on its machine is slightly behind, as shown in Figure 1.Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources. It will probably crash and the programmer will go crazy trying to understand what is wrong with the code.

There are many more examples where an accurate account of time is needed. The example above can easily be reformulated to file timestamps in general. In addition, think of application domains such as financial brokerage, security auditing, and collaborative sensing, and it will become clear that accurate timing is important. Since time is so basic to the way people think and the effect of not having all the clocks synchronized can be so dramatic, it is fitting that we begin our study of synchronization with the simple question: Is it possible to synchronize all the clocks in a distributed system? The answer is surprisingly complicated.

Physical clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word “clock” to refer to these devices, they are not actually clocks in the usual sense. **Timer** is perhaps a better word. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kindof crystal, how it is cut, and the amount of tension. Associated with each crystal are two registers, a **counter** and a **holding register**. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register.In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one **clocktick**.

When the system is booted, it usually asks the user to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory. Most computers have a special battery-backed up CMOS RAM so that the date and time need not be entered on subsequent boots. At every clock tick, the interrupt service procedure adds one to the time stored in memory. In this way, the (software) clock is kept up to date.

With a single computer and a single clock, it does not matter much if thisclock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent. For example, if the file input.c has time 2151 and file input.o has time 2150, make will recompile the source file, even if the clock is off by 2 and the true times are 2153 and 2152,respectively. All that really matters are the relative times[32].

As soon as multiple CPUs are introduced, each with its own clock, the sit-uation changes radically. Although the frequency at which a crystal oscillatorruns is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has n computers, all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of sync and give different values when read out. This difference in time values is called **clock skew**. Asa consequence of this clock skew, programs that expect the time associated with a file, object, process, or message to be correct and independent of the machine on which it was generated (i.e., which clock it used) can fail, as we saw in the makeexample above.

Clock Synchronization Algorithms

If one machine has a UTC receiver, the goal becomes keeping all the othermachines synchronized to it. If no machines have UTC receivers, each machine keeps track of its own time, and the goal is to keep all the machines togetheras well as possible. Many algorithms have been proposed for doing this synchronization.

All clocks are based on some harmonic oscillator: an object that resonates at a certain frequency and from which we can subsequently derive time. Atomic clocks are based on the transitions of the cesium 133 atom, which is not only very high, but also very constant. Hardware clocks in most computers use a crystal oscillator based on quartz, which is also capable of producing a very high, stable frequency, although not as stable as that of atomic clocks. A software clock in a computer is derived from that computer's hardware clock. In particular, the hardware clock is assumed to cause an interrupt f times per second. When this timer goes off, the interrupt handler adds 1 to acounter that keeps track of the number of ticks (interrupts) since some agreed- upon time in the past. This counter acts as a software clock C, resonating at frequency F.

When the UTC time is t, denote by $C_p(t)$ the value of the software clock on machine p. The goal of clock synchronization algorithms is to keep the deviation between the respective clocks of any two machines in a distributedsystem, within a specified bound, known as the **precision** π:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

Note that precision refers to the deviation of clocks only *between machines* that are part of a distributed system. When considering an external reference point, like UTC, we speak of **accuracy**, aiming to keep it bound to a value α:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

The whole idea of clock synchronization is that we keep clocks *precise*, referred to as **internal synchronization** or *accurate*, known as **external synchronization**. A set of clocks that are accurate within bound α, will be precise within bound $\pi = 2\alpha$. However, being precise does not allow us to conclude anything about the accuracy of clocks.

In a perfect world, we would have $C_p(t) = t$ for all p and all t, and thus $\alpha = \pi = 0$. Unfortunately, hardware clocks, and thus also software clocks, are subject to **clock drift**: because their frequency is not perfect and affected by external sources such as temperature, clocks on different machines will gradually start showing different values for time. This is known as the **clock drift rate**: the difference per unit of time from a perfect reference clock. A typical quartz-based hardware clock has a clock drift rate of some 10^{-6} seconds per second, or approximately 31.5 seconds per year. Computerhardware clocks exist that have much lower drift rates.

The specifications of a hardware clock include its **maximum clock drift rate** ρ. If $F(t)$ denotes the actual oscillator frequency of the hardware clock at time t and F its ideal (constant) frequency, then a hardware clock is living up

to its specifications if

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

By using hardware interrupts we are directly coupling a software clock to thehardware clock, and thus also its clock drift rate. In particular, we have that

$$C_p(t) = \frac{1}{F} \int_0^t F(t)dt, \text{ and thus: } \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

which brings us to our ultimate goal, namely keeping the software clock driftrate also bounded to ρ:

$$\forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

If two clocks are drifting from UTC in the opposite direction, at a time Δt after they were synchronized, they may be as much as $2\rho\ \Delta t$ apart. If the system designers want to guarantee a precision π, that is, that no two clocks ever differ by more than π seconds, clocks must be resynchronized (in software) at least every $\pi/(2\rho)$ seconds. The various algorithms differ in precisely how this resynchronization is done.

Network Time Protocol

A common approach in many protocols and originally proposed by Cristian [1989], is to let clients contact a time server. The latter can accurately provide the current time, for example, because it is equipped with a UTC receiver or an accurate clock. The problem, of course, is that when contacting the server, message delays will have outdated the reported time.

In this case, will send a request to B, timestamped with value T_1. B, in turn, will record the time of receipt T_2 (taken from its own local clock), and returns a response timestamped with value T_3, and piggybacking the previously recorded value T_2. Finally, A records the time of the response's arrival, T_4. Let us assume that the propagation delays from A to B is roughly the same as B to A, meaning that

$$\delta T_{req} = T_2 \quad T_1 \quad T_4 \quad T_3 = \delta T_{res}.$$

In that case, A can estimate its offset relative to B as

$$\theta = T_3 + (T_2 - T_1) + (T_4 - T_3) - T_4 = (T_2 - T_1) + (T_3 - T_4)$$

Of course, time is not allowed to run backward. If A's clock is fast, $\theta < 0$, meaning that A should, in principle, set its clock backward. This is not allowed as it could cause serious problems such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change.

Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10 msec to the time. When slowing down, the interrupt routine adds only 9 msec each time until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

In the case of the network time protocol (NTP), this protocol is set up pairwise between servers. In other words, B will also probe A for its current time. The offset θ is computed as given above, along with the estimation δ for the delay:

$$\delta = (T_4 - T_1) - (T_3 - T_2)$$

Eight pairs of (θ, δ) values are buffered, finally taking the minimal value found for δ as the best estimation for the delay between the two servers, and subsequently the associated value θ as the most reliable estimation of the offset.

Applying NTP symmetrically should, in principle, also let B adjust its clock to that of A. However, if B's clock is known to be more accurate, then such an adjustment would be foolish. To solve this problem, NTP divides servers into strata. A server with a reference clock such as a UTC receiver or an atomic clock, is known to be a stratum-1 server (the clock itself is said to operate at stratum 0). When A contacts B, it will adjust only its time if its own stratum level is higher than that of B. Moreover, after the synchronization, A's stratum level will become one higher than that of B. In other words, if B is a stratum-k server, then A will become

a stratum-(k + 1) server if its original stratum level was already larger than k. Due to the symmetry of NTP, if A's stratum level was *lower* than that of B, B will adjust itself to A. There are many important features about NTP, of which many relate to identifying and masking errors, but also security attacks. NTP was originallydescribed in and is known to achieve (worldwide) accuracy in the range of 1–50 msec [33].

Mutual Exclusion

Fundamental to distributed systems is the concurrency and collaboration among multiple processes. In many cases, this also means that processes will need to simultaneously access the same resources. To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes. In this section, we take a look at some important and representative distributed algorithms that have been proposed.

A Centralized Algorithm

A straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator. Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission. If no other process is currently accessing that resource, the coordinator sends back a reply granting permission. When the reply arrives, the requester can go ahead.

Now suppose that another process, P2 in asks for permission to access the resource. The coordinator knows that a different process is already at the resource, so it cannot grant permission. The exact method used to deny permission is system dependent. In the coordinator just refrains from replying, thus blocking process P2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from P2 for the time being and waits for more messages.

The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and accesses the resource. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can go ahead as well.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator lets only one process at a time access the resource. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever no starvation. The scheme is easy to implement, too, and requires only three messages per use of resource request, grant, release. Its simplicity makes it an attractive solution for many practical situations.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck. Nevertheless, the benefits coming from its simplicity outweigh in many cases the potential drawbacks.

A Distributed Algorithm

Using Lamport's logical clocks, and inspired by Lamport's original solution for distributed mutual exclusion which we discussed provided the following algorithm. Their solution

requires a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first. The algorithm works as follows. When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, no message is lost. When a process receives a request message from another process, the action it takes depends on its own state with respect to the resource named in the message. Three different cases have to be clearly distinguished:

If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message has a lower timestamp, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

Process P_0 sends everyone a request with timestamp 8, while at the same time, process P_2 sends everyone a request with timestamp 12. P_1 is not interested in the resource, so it sends OK to both senders. Processes P_0 and P_2 both see the conflict and compare timestamps. P_2 sees that it has lost, so it grants permission to P_0 by sending OK. Process P_0 now queues the request from P_2 for later processing and accesses the resource, as shown in Figure 60(b). When it is finished, it removes the request from P_2 from its queue and sends an OK message to P_2, allowing the latter to go ahead, as shown in Figure 60(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Unfortunately, this algorithm has *N* points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter any of their respective critical regions. The algorithm can be patched up as follows. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent OK message.

Another problem with this algorithm is that either a multicast communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships. Finally, note that *all* processes are involved in *all* decisions concerning accessing the shared resource, which may impose a burden on processes running on resource-constrained machines.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone is overkill. All that is needed is a method to prevent two processes from accessing the resource at the same time. The algorithm can be modified to grant permission when it has collected permission from a simple majority of the other processes, rather than from all of them [34].

A Token Ring Algorithm

A completely different approach to deterministically achieving mutual ex- clusion in a distributed system. In software, we construct an overlay network in the form of a logical ring in which each process is assigned a position in the ring. All that matters is that each processknows who is next in line after itself. When the ring is initialized, process P_0 is given a **token**. The token circulates around the ring. Assuming there are *N* processes, the token is passed from process P_k to process $P_{(k+1) \bmod N}$ in point-to-point messages.

When a process acquires the token from its neighbor, it checks to see if itneeds to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources. After it has finished, it passesthe token along the ring. It is not permitted to immediately enter the resource again using the same token. If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along. As a consequence, when no processes need the resource, the token just circulates around the ring. The correctness of this algorithm is easy to see. Only one process has thetoken at any instant, so only one process can actually get to the resource. Since the token circulates among the processes in a well-defined order, starvationcannot occur. Once a process decides it wants to have access to the resource,at worst it will have to wait for every other process to use the resource.

This algorithm has its own problems. If the token is ever lost, for example, because its holder crashes or due to a lost message containing the token, it must be regenerated. In fact, detecting that it is lost may be difficult, since the amount of time between successive appearances of the token on the networkis unbounded. The fact that the token has not been spotted for an hour doesnot mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is relatively easy. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it thetoken and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

A Decentralized Algorithm

Let us take a look at fully decentralized solution. Each resource is assumed to be replicated *N* times. Every replica has its own coordinator for controlling the access by concurrent processes. However, whenever a process wants to access the resource, it will simplyneed to get a majority vote from $m > N/2$ coordinators. We assume that when a coordinator does not give permission to access a resource which it will do when it had granted permission to another process, it will tell the requester.

The assumption is that when a coordinator crashes, it recovers quickly butwill have forgotten any vote it gave before it crashed. Another way ofviewing this is that a coordinator resets itself at arbitrary moments. The risk that we are taking is that a reset will make the coordinator forget that it had previously granted permission to some process to access the resource. As a consequence, it may incorrectly grant this permission again to another process after its recovery.

Election Algorithms

Many distributed algorithms require one process to act as coordinator, initiator,or otherwise perform some special role. In general, it does not matter which process takes on this special

responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator. If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process P has a unique identifier id(P). In general, election algorithms attempt to locate the process with the highest identifier and designate it as coordinator. The algorithms differ in the way they locate the coordinator.

Furthermore, we also assume that every process knows the identifier of every other process. In other words, each process has complete knowledge of the process group in which a coordinator must be elected. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

The bully algorithm

A well-known solution for electing a coordinator is the **bully algorithm** de- vised by Garcia-Molina [1982]. In the following, we consider N processes $P_0, \ldots, P_{N-1}$ and let $id(P_k) = k$. When any process notices that the coordi- nator is no longer responding to requests, it initiates an election. A process, P_k, holds an election as follows:

a. P_k sends an ELECTION message to all processes with higher identifiers:

   $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.

b. If no one responds, P_k wins the election and becomes coordinator.

If one of the higher-ups answers, it takes over and P_k's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

The group consists of eight processes, with identifiers numbered from 0 to 7. Previously process P_7 was the coordinator, but it has just crashed. Process P_4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely P_5, P_6, and P_7, as shown in Figure 5(a). Processes P_5 and P_6 both respond with OK, as shown in Figure 5(b). Upon getting the first of these responses, P_4 knows that its job is over, knowing that either one of P_5 or P_6 will take over and become coordinator. Process P_4 just sits back and waits to see who the winner will be although at this point it can make a pretty good guess.

In both P_5 and P_6 hold elections, each one sending messages only to those processes with identifiers higher than itself. At this point P_6 knows that P_7 is dead and that it (P_6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, P_6 must now do what is needed. When it is ready to take over,

it announces the takeover by sending a COORDINATOR message to all running processes. When P_4 gets this message, it can now continue with the operation it was trying to do when it discovered that P_7 was dead, but using P_6 as the coordinator this time. In this way the failure of P_7 is handled and the work can continue.

CHAPTER 22

A RING ALGORITHM

Ajay Rastogi, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India
Email Id- ajayrahi@gmail.com

Consider the following election algorithm that is based on the use of a (logical) ring. Unlike some ring algorithms, this one does not use a token. We assume that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process identifier and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step along the way, the sender adds its own identifier to the list in the message effectively making itself a candidate to be elected as coordinator.

Eventually, the message gets back to the process that started it all. That pro- cess recognizes this event when it receives an incoming message containing its own identifier. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest identifier) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

We see what happens if two processes, P_3 and P_6, discover simultaneously that the previous coordinator, process P_7, and has crashed. Each of these builds an ELECTION message and each of them starts circulating its message, independent of the other one. Eventually, both messages will go all the way around, and both P_3 and P_6 will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this is not considered wasteful [35].

Elections in Wireless Environments

Traditional election algorithms are generally based on assumptions that are not realistic in wireless environments. For example, they assume that message passing is reliable and that the topology of the network does not change. These assumptions are false in most wireless environments, especially those for mobile ad hoc networks.

Only few protocols for elections have been developed that work in ad hoc networks. A solution that can handle failing nodes and partitioning networks. An important property of their solution is that the *best* leader can be elected rather than just a random one as was more or less the case in the previously discussed solutions. Their protocol works as follows. To simplify our discussion, we concentrate only on ad hoc networks and ignore that nodes can move.

Consider a wireless ad hoc network. To elect a leader, any node in the network, called the source, can initiate an election by sending an ELECTION message to its immediate neighbors (i.e., the nodes in its range). When a node receives an ELECTION for the first time, it designates the sender as its parent, and subsequently sends out an ELECTION message

to all its immediate neighbors, except for the parent. When a node receives an ELECTION message from a node other than its parent, it merely acknowledges the receipt.

When node R has designated node Q as its parent, it forwards the ELECTION message to its immediate neighbors (excluding Q) and waits for acknowledgments to come in before acknowledging the ELECTION message from Q. This waiting has an important consequence. First, note that neighbors that have already selected a parent will immediately respond to R. More specifically, if all neighbors already have a parent, R is a leaf node and will be able to report back to Q quickly. In doing so, it will also report information such as its battery lifetime and other resource capacities.

This information will later allow Q to compare R's capacities to that of other downstream nodes, and select the best eligible node for leadership. Of course, Q had sent an ELECTION message only because its own parent P had done so as well. In turn, when Q eventually acknowledges the ELECTION message previously sent by P, it will pass the most eligible node to P as well. In this way, the source will eventually get to know which node is best to be selected as leader, after which it will broadcast this information to all other nodes.

When multiple elections are initiated, each node will decide to join only one election. To this end, each source tags its ELECTION message with a unique identifier. Nodes will participate only in the election with the highest identifier, stopping any running participation in other elections. With some minor adjustments, the protocol can be shown to operate also when the network partitions, and when nodes join and leave.

Elections in large-scale systems

Many leader-election algorithms apply to only relatively small distributed systems. Moreover, algorithms often concentrate on the selection of only a single node. There are situations when several nodes should actually be selected, such as in the case of **super peers** in peer-to-peer networks. In this section, we concentrate specifically on the problem of selecting super peers.

The following requirements need to be met for super-peer selection:

A. Normal nodes should have low-latency access to super peers.

B. Super peers should be evenly distributed across the overlay network.

C. There should be a predefined portion of super peers relative to the total number of nodes in the overlay network.

D. Each super peer should not need to serve more than a fixed number of normal nodes.

Fortunately, these requirements are relatively easy to meet in most peer-to- peer systems, given the fact that the overlay network is either structured as in DHT-based systems, or randomly unstructured as, for example, can be realized with gossip-based solutions. In the case of DHT-based systems, the basic idea is to reserve a fraction of the identifier space for super peers. In a DHT-based system, each node receives a random and uniformly assigned m-bit identifier. Now suppose we reserve the first (i.e., leftmost) k bits to identify super peers. For example, if we need N superpeers, then the first $log_2(N)$ bits of any *key* can be used to identify these nodes.

To explain, assume we have a (small) Chord system with $m = 8$ and $k = 3$. When looking up the node responsible for a specific key K, we can first decide to route the lookup

request to the node responsible for the pattern K 11100000 which is then treated as the superpeer.[1] Note that each node with identifier ID can check whether it is a super peer by looking up ID 11100000 to see if this request is routed to itself. Provided node identifiers are uniformly assigned to nodes, it can be seen that with a total of N nodes the number of super peers is, on average, equal to $2^{k-m} N$.

A completely different approach is based on positioning nodes in an m- dimensional geometric space. In this case, assume we need to place N super peers *evenly* throughout the overlay. The basic idea is simple: a total of N tokens are spread across N randomly chosen nodes. No node can hold more than one token. Each token represents a repelling force by which another token is inclined to move away. The net effect is that if all tokens exert the same repulsion force, they will move away from each other and spread themselves evenly in the geometric space.

This approach requires that nodes holding a token learn about other tokens. To this end, we can use a gossiping protocol by which a token's forceis disseminated throughout the network. If a node discovers that the total forces that are acting on it exceed a threshold, it will move the token in the direction of the combined forces. When a token is held by a node for a given amount of time, that node will promote itself to super-peer.

Location systems

When looking at very large distributed systems that are dispersed across a wide-area network, it is often necessary to take proximity into account. Just imagine a distributed system organized as an overlay network in which two processes are neighbors in the overlay network, but are actually placed far apart in the underlying network. If these two processes communicate a lot, itmay have been better to ensure that they are also physically placed in each other's proximity. In this section, we take a look at location-based techniques to coordinate the placement of processes and their communication.

GPS: Global Positioning System

Let us start by considering how to determine your geographical position anywhere on Earth. This positioning problem is by itself solved through a highly specific, dedicated distributed system, namely **GPS**, which is an acronym for **Global Positioning System**. GPS is a satellite-based distributed system that was launched in 1978. Although it initially was used mainly for military applications, it by now has found its way too many civilian applications, notably for traffic navigation. However, many more application domains exist. For example, modern smartphones now allow owners to track each other's position. This principle can easily be applied to tracking other things as well, including pets, children, cars, boats, and so on.

GPS uses up to 72 satellites each circulating in an orbit at a height of approximately 20,000 km. Each satellite has up to four atomic clocks, which are regularly calibrated from special stations on Earth. A satellite continuously broadcasts its position, and time stamps each message with its local time. This broadcasting allows every receiver on Earth to accurately compute its own position using, in principle, only four satellites. To explain, let us first assume that all clocks, including the receiver's, are synchronized.

In order to compute a position, consider first the two-dimensional case, in which three satellites are drawn, along with the circles representing points at the same distance from each respective satellite. We see that the intersection of the three circles is a unique point.

This principle of intersecting circles can be expanded to three dimensions, meaning that we need to know the distance to four satellites to determine the longitude, latitude, and altitude of a receiver on Earth. This positioning is all fairly straightforward, but determining the distance to a satellite becomes complicated when we move from theory to practice. There are at least two important real-world facts that we need to take into account:

i. It takes a while before data on a satellite's position reaches the receiver.

ii. The receiver's clock is generally not in sync with that of a satellite.

Assume that the timestamp from a satellite is completely accurate. Let Δ_r denote the deviation of the receiver's clock from the actual time. When a message is received from satellite S_i with timestamp T_i, then the measured delay Δ_i by the receiver consists of two components: the actual delay, alongwith its own deviation:

$$\Delta_i = (T_{now} - T_i) + \Delta_r$$

where T_{now} is the actual current time. As signals travel with the speed of light, c, the *measured* distance $\tilde{d}_i$ to satellite S_i is equal to $c \cdot \Delta_i$. With

$$d_i = c \cdot (T_{now} - T_i)$$

being the real distance between the receiver and satellite S_i, the measured distance can be rewritten to $\tilde{d}_i = d_i + c \cdot \Delta_r$. The real distance is now computed

$$\tilde{d}_i - c \cdot \Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

where x_i, y_i, and z_i denote the coordinates of satellite S_i. What we see now is a system of quadratic equations with four unknowns (x_r, y_r, z_r, and also Δ_r).We thus need four reference points (i.e., satellites) to find a unique solution that will also give us Δ_r. A GPS measurement will thus also give an account of the actual time.

So far, we have assumed that measurements are perfectly accurate. Of course, they are not. There are many sources of errors, starting with the fact that the atomic clocks in the satellites are not always in perfect sync, theposition of a satellite is not known precisely, the receiver's clock has a finite accuracy, the signal propagation speed is not constant (as signals appear to slow down when entering, e.g., the ionosphere), and so on. On average, this leads to an error of some 5–10 meters. Special modulation techniques, as well as special receivers, are needed to improve accuracy. Using so-called **differential GPS**, by which corrective information is sent through wide-area links, accuracy can be further improved.

When GPS is Not an Option

A major drawback of GPS is that it can generally not be used indoors. For thatpurpose, other techniques are necessary. An increasingly popular technique is to make use of the numerous WiFi access points available. The basic idea is simple: if we have a database of known access points along with their coordinates, and we can estimate our distance to an access point, then with only three detected access points, we should be able to compute our position.Of course, it really is not that simple at all.

A major problem is determining the coordinates of an access point. A popular approach is to do this through war driving: using a WiFi-enabled device along with a GPS receiver, someone drives or walks through an area and records observed access points. An access point

can be identified through its SSID or its MAC-level network address. Accuracy can be improved by taking the observed signal strength into account, and giving more weight to a location with relatively high observed signal strength than to a location where only a weak signal was detected. In the end, we obtain an estimation of the coordinates of the access point. The accuracyof this estimation is strongly influenced by:

1. The accuracy of each GPS detection point $\vec{x}_i$
2. The fact that an access point has a non-uniform transmission range
3. The number of sampled detection points *N*.

Studies show that estimates of the coordinates of an access point may be tens of meters off from the actual location. Moreover, access points come and go at a relatively high rate. Nevertheless, locating and positioning access points is widely popular, exemplified by the open-access Wigle database which is populated through crowd sourcing.

Logical Positioning of Nodes

Instead of trying to find the absolute location of a node in a distributed system, an alternative is to use a logical, proximity-based location. In geometric overlay networks each node is given a position in an *m*-dimensional geometric space, such that the distance between two nodes in that space reflects a real- world performance metric. Computing such a position is the core business of a Network Coordinates System, or simply NCS, which are surveyed by Donnet et al. [2010]. The simplest, and most applied example, is where distance corresponds to internode latency. In other words, given two nodes P and Q, then the distance $\hat{d}(P, Q)$ reflects how long it would take for a message to travel from P to Q and *vice versa*. We use the notation $\hat{d}$ to denote distance in a system where nodes have been assigned coordinates.

There are many applications of geometric overlay networks. Consider the situation where a Web site at server O has been replicated to multiple servers $S_1, \ldots, S_N$ on the Internet. When a client C requests a page from O, the latter may decide to redirect that request to the server closest to C, that is, the one that will give the best response time. If the geometric location of C is known, as well as those of each replica server, O can then simply pick that server S_i for which $\hat{d}(C, S_i)$ is minimal. Note that such a selection requires only local processing at O. In other words, there is, for example, no need to sample all the latencies between C and each of the replica servers.

Another example is optimal replica placement. Consider again a Web site that has gathered the positions of its clients. If the site were to replicate its content to *N* servers, it can compute the *N* best positions where to place replicas such that the average client-to-replica response time is minimal. Performing such computations is almost trivially feasible if clients and servers have geometric positions that reflect internode latencies.

As a last example, consider position-based routing (see, e.g., [Popescu et al., 2012] or [Bilal et al., 2013]). In such schemes, a message is forwarded to its destination using only positioning information. For example, a naive routing algorithm to let each node forward a message to the neighbor closestto the destination. Although it can be easily shown that this specific algorithm need not converge, it illustrates that only local information is used to take a decision. There is no need to propagate link information or such to all nodes in the network, as is the case with conventional routing algorithm

CHAPTER 23

AN ANALYSIS OF CONSISTENCY AND REPLICATION

Manish Joshi, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University,
Moradabad, Uttar Pradesh, India
Email Id- gothroughmanish@gmail.com

An important issue in distributed systems is the replication of data. Dataare generally replicated to enhance reliability or improve performance. Oneof the major problems is keeping replicas consistent. Informally, this means that when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same. In this chapter, we take a detailed look at what consistency of replicated data actually means and the various ways that consistency can be achieved.

We start with a general introduction discussing why replication is useful and how it relates to scalability. We then continue by focusing on what consistency actually means. An important class of what are known as consistency models assumes that multiple processes simultaneously access shared data. Consistency for these situations can be formulated with respect to what processes can expect when reading and updating the shared data, knowing that others are accessing that data as well.

Consistency models for shared data are often hard to implement efficiently in large-scale distributed systems. Moreover, in many cases simpler models can be used, which are also often easier to implement. One specific class is formed by client-centric consistency models, which concentrate on consistency from the perspective of a single (possibly mobile) client. Client-centricconsistency models are discussed in a separate section.

Consistency is only half of the story. We also need to consider how consistency is actually implemented. There are essentially two, more or less independent, issues we need to consider. First of all, we start with concentrating on managing replicas, which takes into account not only the placement of replica servers, but also how content is distributed to these servers.

The second issue is how replicas are kept consistent. In most cases, applications require a strong form of consistency. Informally, this means that updates are to be propagated more or less immediately between replicas. There are various alternatives for implementing strong consistency, which are discussed in a separate section. Also, attention is paid to caching protocols, which form a special case of consistency protocols.

Being arguably the largest distributed system, we pay separate attention to caching and replication in Web-based systems, notably looking at contentdelivery networks as well as edge-server caching techniques[36].

Introduction

In this section, we start with discussing the important reasons for wanting to replicate data in the first place. We concentrate on replication as a technique for achieving scalability, and motivate why reasoning about consistency is so important.

Reasons for replication

There are two primary reasons for replicating data. First, data are replicated to increase the reliability of a system. If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas. Also, by maintaining multiple copies, it becomes possible to provide better protection against corrupted data. For example, imagine there are three copies of a file and every read and write operation is performed on each copy. We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

The other reason for replicating data is performance. Replication for performance is important when a distributed system needs to scale in terms of size or in terms of the geographical area it covers. Scaling with respect to size occurs, for example, when an increasing number of processes needs to access data that are managed by a single server. In that case, performance can be improved by replicating the server and subsequently dividing the workload among the processes accessing the data.

Scaling with respect to a geographical area may also require replication. The basic idea is that by placing a copy of data in proximity of the process using them, the time to access the data decreases. As a consequence, the performance as perceived by that process increases. This example also illustrates that the benefits of replication for performance may be hard to evaluate. Although a client process may perceive better performance, it may also be the case that more network bandwidth is now consumed keeping all replicas up to date.

If replication helps to improve reliability and performance, who could be against it? Unfortunately, there is a price to be paid when data are replicated. The problem with replication is that having multiple copies may lead to consistency problems. Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. Exactly when and how those modifications need to be carried out determines the price of replication.

To understand the problem, consider improving access times to Web pages. If no special measures are taken, fetching a page from a remote Web server may sometimes even take seconds to complete. To improve performance, Web browsers often locally store a copy of a previously fetched Web page (i.e., they *cache* a Web page). If a user requires that page again, the browser automatically returns the local copy. The access time as perceived by the user is excellent. However, if the user always wants to have the latest version of a page, he may be in for bad luck. The problem is that if the page has been modified in the meantime, modifications will not have been propagated to cached copies, making those copies out-of-date.

One solution to the problem of returning a stale copy to the user is to forbid the browser to keep local copies in the first place, effectively letting the server be fully in charge of replication. However, this solution may still lead to poor access times if no replica is placed near the user. Another solution is to let the Web server invalidate or update each cached copy, but this requires that the server keeps track of all caches and sending those messages. This, in turn, may degrade the overall performance of the server. We return to performance versus scalability issues below:

Replication as scaling technique

Replication and caching for performance are widely applied as scaling techniques. Scalability issues generally appear in the form of performance problems. Placing copies of data close to the processes using them can improve performance through reduction of access time and thus solve scalability problems.

A possible trade-off that needs to be made is that keeping copies up to date may require more network bandwidth. Consider a process P that accesses a local replica N times per second, whereas the replica itself is updated M times per second. Assume that an update completely refreshes the previous version of the local replica. If N M, that is, the access-to-update ratio is very low, we have the situation where many updated versions of the local replica will never be accessed by P, rendering the network communication for those versions useless. In this case, it may have been better not to install a local replica close to P, or to apply a different strategy for updating the replica.

A more serious problem, however, is that keeping multiple copies consistent may itself be subject to serious scalability problems. Intuitively, a collection of copies is consistent when the copies are always the same. This means that a read operation performed at any copy will always return the same result. Consequently, when an update operation is performed on one copy, the update should be propagated to all copies before a subsequent operation takes place, no matter at which copy that operation is initiated or performed.

This type of consistency is sometimes informally (and imprecisely) referred to as tight consistency as provided by what is also called synchronous replication. The key idea is that an update is performed at all copies as a single atomic operation, or transaction. Unfor tunately, implementing atomicity involving a large number of replicas that may be widely dispersed across a large-scale network is inherently difficult when operations are also required to complete quickly.

Difficulties come from the fact that we need to synchronize all replicas. In essence, this means that all replicas first need to reach agreement on when exactly an update is to be performed locally. For example, replicas may need to decide on a global ordering of operations using Lamport timestamps, or let a coordinator assign such an order. Global synchronization simply takes a lot of communication time, especially when replicas are spread across a wide-area network.

We are now faced with a dilemma. On the one hand, scalability problems can be alleviated by applying replication and caching, leading to improved performance. On the other hand, to keep all copies consistent generally requires global synchronization, which is inherently costly in terms of performance. The cure may be worse than the disease.

In many cases, the only real solution is to relax the consistency constraints. In other words, if we can relax the requirement that updates need to be executed as atomic operations, we may be able to avoid (instantaneous) global synchronizations, and may thus gain performance. The price paid is that copies may not always be the same everywhere. As it turns out, to what extent consistency can be relaxed depends highly on the access and update patterns of the replicated data, as well as on the purpose for which those data are used[37].

Data-centric consistency models

Traditionally, consistency has been discussed in the context of read and write operations on shared data, available by means of distributed shared memory, a distributed shared database, or a (distributed) file system. Here, we use the broader term **data store**. A data store may

be physically distributed across multiple machines. In particular, each process that can access data from the store is assumed to have a local or nearby copy available of the entire store. Write operations are propagated to the other copies, as shown in Figure 67. A data operation is classified as a write operation when it changes the data, and is otherwise classified as a read operation.

A **consistency model** is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly. Normally, a process that performs a read operationon a data item, expects the operation to return a value that shows the resultsof the last write operation on that data.

In the absence of a global clock, it is difficult to define precisely which write operation is the last one. As an alternative, we need to provide other definitions, leading to a range of consistency models. Each model effectively restricts the values that a read operation on a data item can return. As is to beexpected, the ones with major restrictions are easy to use, for example when developing applications, whereas those with minor restrictions are generallyconsidered to be difficult to use in practice. The trade-off is, of course, that the easy-to-use models do not perform nearly as well as the difficult ones. Such is life.

Continuous Consistency

There is no such thing as a best solution to replicating data. Replicating data poses consistency problems that cannot be solved efficiently in a general way. Only if we loosen consistency can there be hope for attaining efficient solutions. Unfortunately, there are also no general rules for loosening consistency: exactly what can be tolerated is highly dependent on applications.

There are different ways for applications to specify what inconsistencies they can tolerate. Yu and Vahdat [2002] take a general approach by distin- guishing three independent axes for defining inconsistencies: deviation in numerical values between replicas, deviation in staleness between replicas and deviation with respect to the ordering of update operations. They refer to these deviations as forming **continuous consistency** ranges.

Measuring inconsistency in terms of numerical deviations can be usedby applications for which the data have numerical semantics. One obvious example is the replication of records containing stock market prices. In this case, an application may specify that two copies should not deviate more than $0.02, which would be an *absolute numerical deviation*. Alternatively, a *relative numerical deviation* could be specified, stating that two copies should differ by no more than, for example, 0.5%. In both cases, we would see that if a stock goes up (and one of the replicas is immediately updated) without violating the specified numerical deviations, replicas would still be considered to be mutually consistent.

Numerical deviation can also be understood in terms of the number of updates that have been applied to a given replica, but have not yet been seenby others. For example, a Web cache may not have seen a batch of operations carried out by a Web server. In this case, the associated deviation in the *value*is also referred to as its *weight*.

Staleness deviations relate to the last time a replica was updated. For some applications, it can be tolerated that a replica provides old data as long as it is not *too* old. For example, weather reports typically stay reasonably accurate over some time, say a few hours. In such cases, a main server may receive timely updates, but may decide to propagate updates to the replicas only once in a while.

Finally, there are classes of applications in which the ordering of updates are allowed to be different at the various replicas, as long as the differences remain bounded. One way of looking at these updates is that they are applied tentatively to a local copy, awaiting global agreement from all replicas. As a consequence, some updates may need to be rolled back and applied in a different order before becoming permanent. Intuitively, ordering deviations are much harder to grasp than the other two consistency metrics.

The Notion of a Conit

To define inconsistencies, Yu and Vahdat introduce a **consistency unit**, ab- breviated to **conit**. A conit specifies the unit over which consistency is to be measured. For example, in our stock-exchange example, a conit could be defined as a record representing a single stock. Another example is an individual weather report. To give an example of a conit, and at the same time illustrate numerical and ordering deviations, consider the situation of keeping track of a fleet of cars. In particular, the fleet owner is interested in knowing how much he pays on average for gas. To this end, whenever a driver tanks gasoline, he reports the amount of gasoline that has been tanked (recorded as g), the price paid (recorded as p), and the total distance since the last time he tanked (recorded by the variable d). Technically, the three variables g, p, and d form a conit. This conit is replicated across two servers, as shown in Figure 68, and a driver regularly reports his gas usage to one of the servers by separately updating each variable (without further considering the car in question)[38].

The task of the servers is to keep the conit “consistently” replicated. To this end, each replica server maintains a two-dimensional vector clock. We use the notation T, R to express an operation that was carried out by replica R at (its) logical time T.

The numerical deviation at a replica R consists of two components: the number of operations at all *other* replicas that have not yet been seen by R, along with the sum of corresponding missed values (more sophisticated schemes are, of course, also possible). In our example, A has not yet seen operations 6, B and 7, B with a total value of 70 + 412 units, leading to a numerical deviation of (2, 482). Likewise, B is still missing the three tentative operations at A, with a total summed value of 686, bringing B’s numerical deviation to (3, 686).

Using these notions, it becomes possible to specify specific consistency schemes. For example, we may restrict order deviation by specifying an acceptable maximal value. Likewise, we may want two replicas to never numerically deviate by more than 1000 units. Having such consistency schemes does require that a replica knows how much it is deviating from other replicas, implying that we need separate communication to keep replicas informed. The underlying assumption is that such communication is much less expensive than communication to keep replicas synchronized. Admittedly, it is questionable if this assumption also holds for our example.

CHAPTER 24

IMPLEMENTING CLIENT-CENTRIC CONSISTENCY

Namit Gupta, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University,
Moradabad, Uttar Pradesh, India
Email Id- namit.k.gupta@gmail.com

For our last topic on consistency protocols, let us draw our attention to implementing client-centric consistency. Implementing client-centric consistency is relatively straightforward if performance issues are ignored. In a naive implementation of client-centric consistency, each write opera- tion W is assigned a globally unique identifier. Such an identifier is assigned by the server to which the write had been submitted. We refer to this server as the **origin** of W. Then, for each client, we keep track of two sets of writes. The read set for a client consists of the writes relevant for the read operations performed by a client. Likewise, the write set consists of the (identifiers of the) writes performed by the client.

Monotonic-read consistency is implemented as follows. When a client performs a read operation at a server, that server is handed the client's read set to check whether all the identified writes have taken place locally. If not, it contacts the other servers to ensure that it is brought up to date before carrying out the read operation. Alternatively, the read operation is forwarded to a server where the write operations have already taken place. After the read operation is performed, the write operations that have taken place at the selected server and which are relevant for the read operation are added to the client's read set[39].

Note that it should be possible to determine exactly where the write operations identified in the read set have taken place. For example, the write identifier could include the identifier of the server to which the operation was submitted. That server is required to, for example, log the write operation so that it can be replayed at another server. In addition, write operations should be performed in the order they were submitted. Ordering can be achieved by letting the client generate a globally unique sequence number that is included in the write identifier. If each data item can be modified only by its owner, the latter can supply the sequence number.

Monotonic-write consistency is implemented analogous to monotonic reads. Whenever a client initiates a new write operation at a server, the server is handed over the client's write set. Again, the size of the set may be prohibitively large in the face of performance requirements. An alternative solution is discussed below. It then ensures that the identified write operations are performed first and in the correct order. After performing the new operation, that operation's write identifier is added to the write set. Note that bringing the current server up to date with the client's write set may introduce a considerable increase in the client's response time since the client then waits for the operation to fully complete.

Likewise, read-your-writes consistency requires that the server where the read operation is performed has seen all the write operations in the client's write set. The writes can simply be fetched from other servers before the read operation is performed, although this may lead to a poor response time. Alternatively, the client-side software can search for a server where the identified write operations in the client's write set have already been performed.

Finally, writes-follow-reads consistency can be implemented by first bring- ing the selected server up to date with the write operations in the client's read set, and then later adding the

identifier of the write operation to the write set, along with the identifiers in the read set which have now become relevant for the write operation just performed.

For clarity, assume that for each server, writes from S_j are processed in the order that they were submitted. Whenever a client issues a request to perform a read or write operation O at a specific server, that server returns its current timestamp along with the results of O. Read and write sets are subsequently represented by vector timestamps. More specifically, for each session A, we construct a vector timestamp SVC_A with $SVC_A[i]$ set equal to the maximum timestamp of all write operations in A that originate from server S_i:

$$SVC_A[j] = \max\{ts(W) | W \in A \text{ and } origin(W) = S_j\}$$

In other words, the timestamp of a session always represents the latest write operations that have been seen by the applications that are being executed as part of that session. The compactness is obtained by representing all observed write operations originating from the same server through a single timestamp.

As an example, suppose a client, as part of session A, logs in at server S_i. To that end, it passes SVC_A to S_i. Assume that $SVC_A[j] > WVC_i[j]$. What this meansis that S_i has not yet seen all the writes originating from S_j that the client has seen. Depending on the required consistency, server S_i may now have to fetch

these writes before being able to consistently report back to the client. Once the operation has been performed, server S_i will return its current timestamp WVC_i. At that point, SVC_A is adjusted to:

$$SVC_A[j] \leftarrow \max\{SVC_A[j], WVC_i[j]\}$$

Again, we see how vector timestamps can provide an elegant and compact way of representing history in a distributed system.

The Web is arguably the largest distributed system ever built. Originating from a relatively simple client-server architecture, it is now a sophisticated system consisting of many techniques to ensure stringent performance and availability requirements. These requirements have led to numerous proposals for caching and replicating Web content. Where the original schemes (which are still largely deployed) have been targeted toward supporting static content, much effort has also been put into supporting dynamic content, that is, supporting documents that are generated on-the-spot as the result of a request, as well as those containing scripts and such.

Client-side caching in the Web generally occurs at two places. In the first place, most browsers are equipped with a relatively simple caching facility. Whenever a document is fetched it is stored in the browser's cache from where it is loaded the next time. In the second place, a client's site often runs a Webproxy. A Web proxy accepts requests from local clients and passes these to Web servers. When a response comes in, the result is passed to the client. The advantage of this approach is that the proxy can cache the result and return that result to another client, if necessary. In other words, a Web proxy can implement a shared cache. With so many documents being generated on thefly, the server generally provides the document in pieces instructing the client to cache only those parts that are not likely to change when the document is requested a next time.

In addition to caching at browsers and proxies, ISPs generally also place caches in their networks. Such schemes are mainly used to reduce network traffic (which is good for the

ISP) and to improve performance (which is good for end users). However, with multiple caches along the request path from client to server, there is a risk of increased latencies when caches do notcontain the requested information.

As an alternative to building hierarchical caches, one can also organize caches for cooperative deployment. In cooperative caching or distributed caching, whenever a cache miss occurs at a Web proxy, the proxy first checks a number of neighboring proxies to see if one of them contains the requested document. If such a check fails, the proxy forwards the request to the Web server responsible for the document. In more traditional settings, this scheme is primarily deployed with Web caches belonging to the same organization or institution.

As mentioned, there are essentially only three (related) measures that can be taken to change the behavior of a Web hosting service: changing the placement of replicas, changing consistency enforcement, and deciding on how and when to redirect client requests. We already discussed the first two measures extensively. Client-request redirection deserves some more attention.Before we discuss some of the trade-offs, let us first consider how consistency and replication are dealt with in a practical setting by considering the Akamai situation.

The basic idea is that each Web document consists of a main HTML (or XML) page in which several other documents such as images, video, and audio have been embedded. To display the entire document, it is necessary that the embedded documents are fetched by the user's browser as well.The assumption is that these embedded documents rarely change, for whichreason it makes sense to cache or replicate them.

Each embedded document is normally referenced through a URL. However, in Akamai's CDN, such a URL is modified such that it refers to a **virtual ghost**, which is a reference to an actual server in the CDN. The URL also contains the host name of the origin server for reasons we explain next. The name of the virtual ghost includes a DNS name such as ghosting.com, which is resolved by the regular DNS naming system to a CDN DNS server the result of step 3).

Each such DNS server keeps track of servers close to the client. To this end, any of the proximity metrics we have discussedpreviously could be used. In effect, the CDN DNS servers redirect the client to a replica server best for that client (step 4), which could mean the closest one, the least-loaded one, or a combination of several such metrics the actual redirection policy is proprietary.

Finally, the client forwards the request for the embedded document to the selected CDN server. If this server does not yet have the document, it fetches it from the original. If the document was already in the CDN server's cache, it can be returned forthwith. Note that in order to fetch the embedded document, the replica server must be able to send a request to the origin server, for which reason its host name is also contained in the embedded document's URL.

An interesting aspect of this scheme is the simplicity by which consistencyof documents can be enforced. Clearly, whenever a main document is changed,a client will always be able to fetch it from the origin server. In the case of embedded documents, a different approach needs to be followed as these documents are, in principle, fetched from a nearby replica server. To this end,a URL for an embedded document not only refers to a special host name thateventually leads to a CDN DNS server, but also contains a unique identifier that is changed every time the embedded document changes. In effect, this identifier changes the name of the embedded document. As a consequence, when the client is redirected to a

specific CDN server, that server will not find the named document in its cache and will thus fetch it from the origin server. The old document will eventually be evicted from the server's cache as it is no longer referenced.

This example already shows the importance of client-request redirection.In principle, by properly redirecting clients, a CDN can stay in control when it comes to client-perceived performance, but also taking into account global system performance by, for example, avoiding that requests are sent to heavily loaded servers. These so-called **adaptive redirection policies** can be applied when information on the system's current behavior is provided to the processes that take redirection decisions. This brings us partly back to the metric estimation techniques discussed previously[40].

Besides the different policies, an important issue is whether request redirection is transparent to the client or not. In essence, there are only three redirection techniques: TCP handoff, DNS redirection, and HTTP redirection. We already discussed TCP handoff. This technique is applicable only for server clusters and does not scale to wide-area networks.

DNS redirection is a transparent mechanism by which the client can bekept completely unaware of where documents are located. Akamai's two-level redirection is one example of this technique. We can also directly deploy DNS to return one of several addresses as we discussed before. Note, however, that DNS redirection can be applied only to an entire site: the name of individual documents does not fit into the DNS name space.

HTTP redirection, finally, is a nontransparent mechanism. When a client requests a specific document, it may be given an alternative URL as partof an HTTP response message to which it is then redirected. An important observation is that this URL is visible to the client's browser. In fact, the user may decide to bookmark the referral URL, potentially rendering the redirection policy useless.

Up to this point we have mainly concentrated on caching and replicating static Web content. In practice, we see that the Web is increasingly offering more dynamically generated content, but that it is also expanding toward offering services that can be called by remote applications. Also in these situa- tions we see that caching and replication can help considerably in improving the overall performance, although the methods to reach such improvements are more subtle than what we discussed so far.

When considering improving performance of Web applications through caching and replication, matters are complicated by the fact that several solutions can be deployed, with no single one standing out as the best. Let us consider the edge-server situation as sketched. In this case, we assume a CDN in which each hosted site has an origin server that acts as the authoritative site for all read and update operations.

Recall that in an edge-server architecture, Web clients request data through an edge server, which, in turn, gets its information from the origin server associated with the specific Web site referred to by the client. As also shown in Figure 4 we assume that the origin server consists of a database from which responses are dynamically created. Although we have shown only a single Web server, it is common to organize each server according to a multitier architecture as we discussed before. An edge server can now be roughly organized along the following lines.

First, to improve performance, we can decide to apply full replication of the data stored at the origin server. This scheme works well whenever the update ratio is low and when queries require an extensive database search. As mentioned above, we assume that all

updates are carried out at the origin server, which takes responsibility for keeping the replicas and the edge servers in a consistent state. Read operations can thus take place at the edge servers. Here we see that replicating for performance will fail when the update ratio is high, as each update will incur communication over a wide-area network to bring the replicas into a consistent state. The read/update ratio is the determining factor to what extent the origin database in a wide-area setting should be replicated.

Another case for full replication is when queries are generally complex. In the case of a relational database, this means that a query requires that multiple tables need to be searched and processed, as is generally the case with a join operation. Opposed to complex queries are simple ones that generally require access to only a single table in order to produce a response. In the latter case, partial replication by which only a subset of the data is stored at the edge server may suffice.

An alternative to partial replication is to make use of content-aware caches. The basic idea in this case is that an edge server maintains a local database that is now tailored to the type of queries that can be handled at the origin server. To explain, in a full-fledged database system a query will operate on a database in which the data has been organized into tables such that, for example, redundancy is minimized. Such databases are also said to be normalized.

In such databases, any query that adheres to the data schema can, in principle, be processed, although perhaps at considerable costs. With content- aware caches, an edge server maintains a database that is organized according to the structure of queries. What this means is that queries are assumed to adhere to a limited number of templates, effectively meaning that the different kinds of queries that can be processed is restricted. In these cases, whenever a query is received, the edge server matches the query against the available templates, and subsequently looks in its local database to compose a response, if possible. If the requested data is not available, the query is forwarded to the origin server after which the response is cached before returning it to the client.

In effect, what the edge server is doing is checking whether a query can be answered with the data that is stored locally. This is also referred to as a query containment check. Note that such data was stored locally as responses to previously issued queries. This approach works best when queries tend to be repeated. Part of the complexity of content-aware caching comes from the fact that the data at the edge server needs to be kept consistent. To this end, the origin server needs to know which records are associated with which templates, so that any update of a record, or any update of a table, can be properly addressed by, for example, sending an invalidation message to the appropriate edge servers. Another source of complexity comes from the fact that queries still need to be processed at edge servers. In other words, there is no negligible computational power needed to handle queries.

Considering that databases often form a performance bottleneck in Web servers, alternative solutions may be needed. Finally, caching results from queries that span multiple tables (i.e., when queries are complex) such that a query containment check can be carried out effectively is not trivial. The reason is that the organization of the results may be very different from the organization of the tables on which the query operated.

These observations lead us to a third solution, namely content-blind caching. The idea of content-blind caching is extremely simple: when a client submits a query to an edge server, the server first computes a unique hash value for that query. Using this hash value, it subsequently looks in its cache whether it has processed this query before. If not, the query is forwarded to the origin and the result is cached before returning it to the client. If the query had been processed before, the previously cached result is returned to the client.

The main advantage of this scheme is the reduced computational effortthat is required from an edge server in comparison to the database approaches described above. However, content-blind caching can be wasteful in terms of storage as the caches may contain much more redundant data in comparisonto content-aware caching or database replication. Note that such redundancy also complicates the process of keeping the cache up to date as the originserver may need to keep an accurate account of which updates can potentially affect cached query results. These problems can be alleviated when assuming that queries can match only a limited set of predefined templates as we discussed above.

CHAPTER 25

AN INTRODUCTION OF FAULT TOLERANCE

Pradeep Kumar Shah, Assistant Professor
College of Computing Sciences and IT, Teerthanker Mahaveer University,
Moradabad, Uttar Pradesh, India
Email Id- pradeep.rdndj@gmail.com

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure: part of the system is failing while the remaining part continues to operate, and seemingly correctly. An important goal in distributed-systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the system should continue to operate in an acceptable way while repairs are being made. In other words, a distributed system is expected to be fault tolerant.

In this chapter, we take a closer look at techniques to achieve fault tolerance. After providing some general background, we will first look at process resilience through process groups. In this case, multiple identical processes cooperate providing the appearance of a single logical process to ensure thatone or more of them can fail without a client noticing. A specifically difficult point in process groups is reaching consensus among the group memberson which a client-requested operation is to perform. By now, Paxos is a commonly adopted, yet relatively intricate algorithm, which we explain by building it from the ground up. Likewise, we carefully examine the cases in which consensus can be reached, and under which circumstances.

Achieving fault tolerance and reliable communication are strongly related. Next to reliable client-server communication we pay attention to reliable group communication and notably atomic multicasting. In the latter case, a message is delivered to all nonfaulty processes in a group, or to none at all. Having atomic multicasting makes development of fault-tolerant solutions much easier.

Atomicity is a property that is important in many applications. In this chapter, we pay attention to what are known as distributed commit protocols by which a group of processes are conducted to either jointly commit their local work, or collectively abort and return to a previous system state. Finally, we will examine how to recover from a failure. In particular, we consider when and how the state of a distributed system should be saved to allow recovery to that state later on[41].

Introduction to fault tolerance

Fault tolerance has been subject to much research in computer science. In this section, we start with presenting the basic concepts related to processing failures, followed by a discussion of failure models. The key technique for handling failures is redundancy, which is also discussed.

Basic concepts

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults. Being fault tolerant is strongly related to what are called **dependable systems**. Dependability is a term that covers a number of useful requirements for distributed systems including the following:

Availability

It is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.

Reliability

It refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability. If a system goes down on average for one, seemingly random millisecond every hour, it has an availability of more than 99.9999 percent, but is still unreliable. Similarly, a system that never crashes but is shut down for two specific weeks every August has high reliability but only 96 percent availability. The two are not the same.

Safety

It refers to the situation that when a system temporarily fails to operate correctly, no catastrophic event happens. For example, many process- control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous. Many examples from the past (and probably many more yet to come) show how hard it is to build safe systems.

Maintainability

It refers to how easily a failed system can be repaired. A highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically. However, as we shall see later in this chapter, automatically recovering from failures is easier said than done.

Failure models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with one another and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do. However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else.

Such dependency relations appear in abundance in distributed systems. A failing disk may make life difficult for a file server that is designed to provide a highly available file system. If such a file server is part of a distributed database, the proper working of the entire database

may be at stake, as only part of its data may be accessible. To get a better grasp on how serious a failure actually is, several classification schemes have been developed. One such scheme is shown in Table 1.

Table 1: Represented that Different types of failures.

Type of failure	Description of server's behavior
Crash failure	Halts, but is working correctly until it halts
Omission failure Receive omission Send omission	Fails to respond to incoming requests Fails to receive incoming messages Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure Value failure State-transition failure	Response is incorrect The value of the response is wrong Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

A crash failure occurs when a server prematurely halts, but was workingcorrectly until it stopped. An important aspect of crash failures is that once the server has halted, nothing is heard from it anymore. A typical example of a crash failure is an operating system that comes to a grinding halt, and for which there is only one solution: reboot it. Many personal computer systems suffer from crash failures so often that people have come to expect them to be normal. Consequently, moving the reset button from the back of a cabinet tothe front was done for good reason. Perhaps one day it can be moved to the back again, or even removed altogether.

An omission failure occurs when a server fails to respond to a request. Several things might go wrong. In the case of a receive-omission failure, possibly the server never got the request in the first place. Note that it may well be the case that the connection between a client and a server has been correctly established, but that there was no thread listening to incoming requests. Also, a receive-omission failure will generally not affect the current state of the server, as the server is unaware of any message sent to it [42].

Likewise, a send-omission failure happens when the server has done its work, but somehow fails in sending a response. Such a failure may happen, for example, when a send buffer overflows while the server was not preparedfor such a situation. Note that, in contrast to a receive-omission failure, the server may now be in a state reflecting that it has just completed a service forthe client. As a consequence, if the sending of its response fails, the server hasto be prepared for the client to reissue its previous request.

Other types of omission failures not related to communication may be caused by software errors such as infinite loops or improper memory management by which the server is said to "hang."

Another class of failures is related to timing. Timing failures occur when the response lies outside a specified real-time interval. For example, in the case of streaming video's, providing data too soon may easily cause trouble for a recipient if there is not enough buffer space to hold all the incoming data. More common, however, is that a server responds too late, in which case a *performance* failure is said to occur.

A serious type of failure is a response failure, by which the server's response is simply incorrect. Two kinds of response failures may happen. In the case of a value failure, a server simply provides the wrong reply to a request. For example, a search engine that systematically returns Web pages not related to any of the search terms used, has failed.

The other type of response failure is known as a state-transition failure. This kind of failure happens when the server reacts unexpectedly to an incoming request. For example, if a server receives a message it cannot recognize, a state-transition failure happens if no measures have been taken to handle such messages. In particular, a faulty server may incorrectly take default actions it should never have initiated.

The most serious are arbitrary failures, also known as Byzantine failures. In effect, when arbitrary failures occur, clients should be prepared for the worst. In particular, it may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect. We return to such failures below.

Many of the aforementioned cases deal with the situation that a process P no longer perceives any actions from another process Q. However, can P conclude that Q has indeed come to a halt? To answer this question, we need to make a distinction between two types of distributed systems:

1. In an asynchronous system, no assumptions about process execution speeds or message delivery times are made. The consequence is that when process P no longer perceives any actions from Q, it cannot conclude that Q crashed. Instead, it may just be slow or its messages may have been lost.

2. In a synchronous system, process execution speeds and message-delivery times are bounded. This also means that when Q shows no more activity when it is expected to do so, process P can rightfully conclude that Q has crashed.

Unfortunately, pure synchronous systems exist only in theory. On the other hand, simply stating that every distributed system is asynchronous also does not do just to what we see in practice and we would be overly pessimistic in designing distributed systems under the assumption that they are necessarily asynchronous. Instead, it is more realistic to assume that a distributed system is partially synchronous: most of the time it behaves as a synchronous system, yet there is no bound on the time that it behaves in an asynchronous fashion. In other words, asynchronous behavior is an exception, meaning that we can normally use timeouts to conclude that a process has indeed crashed, but that occasionally such a conclusion is false. In practice, this means that we will have to design fault-tolerant solutions that can withstand incorrectly detecting that a process halted [43].

1. Fail-stop failures refer to crash failures that can be reliably detected. This may occur when assuming nonfaulty communication links and when the failure-detecting process P can place a worst-case delay on responses from Q.

2. Fail-noisy failures are like fail-stop failures, except that P will only *eventually* come to the correct conclusion that Q has crashed. This means that there may be some a priori unknown time in which P's detections of the behavior of Q are unreliable.

3. When dealing with fail-silent failures, we assume that communication links are nonfaulty, but that process P cannot distinguish crash failures from omission failures.
4. Fail-safe failures cover the case of dealing with arbitrary failures by process Q, yet these failures are benign: they cannot do any harm.
5. Finally, when dealing with fail-arbitrary failures, Q may fail in any possible way; failures may be unobservable in addition to being harmful to the otherwise correct behavior of other processes.

Clearly, having to deal with fail-arbitrary failures is the worst that can happen. As we shall discuss shortly, we can design distributed systems in such a way that they can even tolerate these types of failures.

Failure masking by redundancy

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy with information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With time redundancy, an action is performed, and then, if need be, it is performed again. Transactions use this approach. If a transaction aborts, it can be redone with no harm. Another well-known example is retransmitting a request to a server when lacking an expected response. Time redundancy is especially helpful when the faults are transient or intermittent [44].

With physical redundancy, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunction-ing of some components. Physical redundancy can thus be done either in hardware or in software. For example, extra processes can be added to the system so that if a small number of they crash, the system can still function correctly. In other words, by replicating processes, a high degree of fault tolerance may be achieved. We return to this type of software redundancy later in this chapter [45].

QUESTIONS FOR REVISION

1. What is the architectural model?
2. What is Interaction Models, Failure models, Security models?
3. What are the types of communication paradigm in DS?
4. What are Software and hardware service layers in distributed systems?
5. What are Difficulties and threats for distributed systems?
6. What is the difference between RMI and RPC?
7. Difference between synchronous and asynchronous communication?
8. What infrastructure provided by multicast message for distributed system?
9. What are the Applications of publish-subscribe systems?
10. What are the three types of communication paradigm in distributed system?
11. Compare the communicating entities: object, components and web services.

REFERENCES

[1] M. van Steen and A. S. Tanenbaum, "A brief introduction to distributed systems," Computing, 2016, doi: 10.1007/s00607-016-0508-7.

[2] N. Vlassis, "A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence," Synth. Lect. Artif. Intell. Mach. Learn., 2007, doi: 10.2200/s00091ed1v01y200705aim002.

[3] D. Brefort et al., "An architectural framework for distributed naval ship systems," Ocean Eng., 2018, doi: 10.1016/j.oceaneng.2017.10.028.

[4] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart, "An Introduction to Distributed Systems," in Web Data Management, 2013. doi: 10.1017/cbo9780511998225.015.

[5] L. Czaja, Introduction to Distributed Computer Systems. 2018.

[6] G. Fortino, G. Di Fatta, M. Pathan, and A. V. Vasilakos, "Cloud-assisted body area networks: state-of-the-art and future challenges," Wirel. Networks, 2014, doi: 10.1007/s11276-014-0714-1.

[7] C. Lin and M. Lin, "The determinants of using cloud supply chain adoption," Ind. Manag. Data Syst., 2019, doi: 10.1108/IMDS-12-2017-0589.

[8] Y. Bounagui, A. Mezrioui, and H. Hafiddi, "Toward a unified framework for Cloud Computing governance: An approach for evaluating and integrating IT management and governance models," Comput. Stand. Interfaces, 2019, doi: 10.1016/j.csi.2018.09.001.

[9] M. Capra, R. Peloso, G. Masera, M. R. Roch, and M. Martina, "Edge computing: A survey on the hardware requirements in the Internet of Things world," Future Internet. 2019. doi: 10.3390/fi11040100.

[10] U. M. Ismail, S. Islam, M. Ouedraogo, and E. Weippl, "A framework for security transparency in Cloud Computing," Futur. Internet, 2016, doi: 10.3390/fi8010005.

[11] U. Jahn, C. Wolff, and P. Schulz, "Concepts of a modular system architecture for distributed robotic systems," Computers, 2019, doi: 10.3390/computers8010025.

[12] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger, "Visualizing request-flow comparison to aid performance diagnosis in distributed systems," IEEE Trans. Vis. Comput. Graph., 2013, doi: 10.1109/TVCG.2013.233.

[13] Q. Rouland, B. Hamid, and J. Jaskolka, "Formal specification and verification of reusable communication models for distributed systems architecture," Futur. Gener. Comput. Syst., 2020, doi: 10.1016/j.future.2020.02.033.

[14] N. Bastin et al., "The InstaGENI initiative: An architecture for distributed systems and advanced programmable networks," Comput. Networks, 2014, doi: 10.1016/j.bjp.2013.12.034.

[15] I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," 2002. doi: 10.1145/582128.582135.

[16] B. Merker, "Consciousness without a cerebral cortex: A challenge for neuroscience and medicine," Behav. Brain Sci., 2007, doi: 10.1017/S0140525X07000891.

[17] G. P. C. Salmond, B. W. Bycroft, G. S. A. B. Stewart, and P. Williams, "The bacterial 'enigma': cracking the code of cell–cell communication," Molecular Microbiology. 1995. doi: 10.1111/j.1365-2958.1995.tb02424.x.

[18] M. Van Der Linden, "Une approche cognitive du fonctionnement de la mémoire épisodique et de la mémoire autobiographique," Clin. Mediterr., 2003, doi: 10.3917/cm.067.0053.

[19] M. C. de F. Sanches, B. H. Ortuño, and S. R. M. Martins, "Fashion Design: The Project of the Intangible," Procedia Manuf., 2015, doi: 10.1016/j.promfg.2015.07.377.

[20] M. M. Mesulam, "From sensation to cognition," Brain. 1998. doi: 10.1093/brain/121.6.1013.

[21] M. A. Bakr and S. Lee, "Distributed multisensor data fusion under unknown correlation and data inconsistency," Sensors (Switzerland). 2017. doi: 10.3390/s17112472.

[22] V. Guleva, E. Shikov, K. Bochenina, S. Kovalchuk, A. Alodjants, and A. Boukhanovsky, "Emerging complexity in distributed intelligent systems," Entropy, 2020, doi: 10.3390/e22121437.

[23] J. Wilson, C. B. Crisp, and M. M. Mortensen, "Extending construal-level theory to distributed groups: Understanding the effects of virtuality," Organ. Sci., 2013, doi: 10.1287/orsc.1120.0750.

[24] T. O. Vuori and Q. N. Huy, "Distributed Attention and Shared Emotions in the Innovation Process: How Nokia Lost the Smartphone Battle," Adm. Sci. Q., 2016, doi: 10.1177/0001839215606951.

[25] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," ACM Comput. Surv., 1989, doi: 10.1145/72551.72552.

[26] P. Sousa, A. Silva, and J. Marques, "Naming and identification in distributed systems: A pattern for naming policies," … Pattern Lang. Programs, …, 1996.

[27] A. Bawden, "Implementing Distributed Systems Using Linear Naming," MIT A.I. Tech. Rep., 1993.

[28] K. Grechuta et al., "Multisensory cueing facilitates naming in aphasia," J. Neuroeng. Rehabil., 2020, doi: 10.1186/s12984-020-00751-w.

[29] S. Zatti, J. Ashfield, J. Baker, and E. Miller, "Naming and registration for IBM distributed systems," IBM Syst. J., 2010, doi: 10.1147/sj.312.0353.

[30] Y. Song, X. He, Z. Liu, W. He, C. Sun, and F. Y. Wang, "Parallel control of distributed parameter systems," IEEE Trans. Cybern., 2018, doi: 10.1109/TCYB.2018.2849569.

[31] T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination," ACM Comput. Surv., 1994, doi: 10.1145/174666.174668.

[32] S. Kitagawa, R. Kitaura, and S. I. Noro, "Functional porous coordination polymers," Angewandte Chemie - International Edition. 2004. doi: 10.1002/anie.200300610.

[33] A. Bigsten and S. Tengstam, "International coordination and the effectiveness of aid," World Dev., 2015, doi: 10.1016/j.worlddev.2013.12.021.

[34] H. Z. Ceballos and J. E. O. Triviño, “S.O.V.O.R.A.: A distributed wireless operating system,” Inf., 2020, doi: 10.3390/info11120581.

[35] J. Wolbers, K. Boersma, and P. Groenewegen, “Introducing a Fragmentation Perspective on Coordination in Crisis Management,” Organ. Stud., 2018, doi: 10.1177/0170840617717095.

[36] G. Francis, “Replication, statistical consistency, and publication bias,” J. Math. Psychol., 2013, doi: 10.1016/j.jmp.2013.02.003.

[37] H. Shen, “IRM: Integrated file replication and consistency maintenance in P2P systems,” IEEE Trans. Parallel Distrib. Syst., 2010, doi: 10.1109/TPDS.2009.43.

[38] J. Shute et al., “F1: A distributed SQL database that scales,” Proc. VLDB Endow., 2013, doi: 10.14778/2536222.2536232.

[39] H. Wang, J. Li, H. Zhang, and Y. Zhou, “Benchmarking replication and consistency strategies in cloud serving databases: HBase and Cassandra,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), 2014, doi: 10.1007/978-3-319-13021-7_6.

[40] D. M. Barch and D. Pagliaccio, “Consistency, replication, and meta-analyses of altered brain activity in unipolar depression,” JAMA Psychiatry. 2017. doi: 10.1001/jamapsychiatry.2016.2844.

[41] A. U. Rehman, R. L. Aguiar, and J. P. Barraca, “Fault-tolerance in the scope of Software-Defined Networking (SDN),” IEEE Access, 2019, doi: 10.1109/ACCESS.2019.2939115.

[42] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, “Debugging distributed systems: Challenges and options for validation and debugging,” Cacm, 2016.

[43] C. J. Goodrum, C. P. F. Shields, and D. J. Singer, “Understanding cascading failures through a vulnerability analysis of interdependent ship-centric distributed systems using networks,” Ocean Eng., 2018, doi: 10.1016/j.oceaneng.2017.12.039.

[44] I. Sinitsyn, A. Mironov, Y. Vorontsov, N. Borzykh, and E. Mikhailova, “The principles of synchronization in the distributed information systems,” Econ. Ann., 2020, doi: 10.21003/EA.V183-08.

[45] M. Mudassar, Y. Zhai, L. Liao, and J. Shen, “A Decentralized Latency-Aware Task Allocation and Group Formation Approach with Fault Tolerance for IoT Applications,” IEEE Access, 2020, doi: 10.1109/ACCESS.2020.2979939.