Distributed and Operating Systems
Course Notes

Prashant Shenoy
University of Massachusetts Amherst

Contents

1 Lecture 1: Introduction **3**

2 Lecture 2: System Architectures **9**

3 Lecture 3: Communication in Distributed Systems **21**

4 Lecture 4: Remote Method Invocation **24**

5 Lecture 5: Concurrency Models **35**

6 Lecture 6: Distributed and Cluster scheduling **40**

7 Lecture 7: Cluster scheduling and Virtualization **51**

8 Lecture 8: OS Virtualization **58**

9 Lecture 9: OS Virtualization and Process Migration **64**

10 Lecture 10: VM/container Migration and Orchestration **69**

11 Lecture 11: Cloud Computing **74**

12 Lecture 12: Clock sychronization and Logical clocks **85**

13 Lecture 13: Vector clocks and distributed snapshots **90**

14 Lecture 14: Leader election, Distributed locks **97**

15 Lecture 15: Distributed transactions **105**

16 Lecture 16: Consistency Models **116**

17 Lecture 17: Replication in distributed systems **127**

18 Lecture 18: Byzantine Fault Tolerance (BFT) **132**

19 Lecture 19: Distributed consensus, paxos, RAFT **143**

20 Lecture 20: Distributed Web Services **153**

21 Lecture 21: Edge computing, web caching **165**

22 Lecture 22: Distributed file systems **173**

23 Lecture 23: xFS, Coda, HDFS, GFS **178**

24 Lecture 24: Distributed middleware **187**

25 Lecture 25: Distributed systems security **196**

26 Lecture 26: Pervasive computing and Internet of Things **208**

CMPSCI 677 Distributed and Operating Systems **Spring 2022**

Lecture 1: January 26

Lecturer: Prashant Shenoy Scribe: Bin Wang(2018), Jonathan Westin(2019), Mehmet Savasci(2022)

1.1 Introduction to the course

The lecture started by outlining logistics (the course web page (`http://lass.cs.umass.edu/~shenoy/courses/677`), course syllabus, staff, textbook, Piazza, YouTube live lecture, topics, and exam schedule) about the course. The instructor recommended "Distributed Systems, 3rd ed" by Tannenbaum and Van Steen as the textbook. It is legally available for free.

The instructor also introduced the grading scheme and other resources. It is worth noting that programming assignments and exams (one midterm and one final exam) contribute heavily towards the final grade. Please note that **no laptop/device use is allowed during the class.**

Two important questions were answered before academic part was started. Here are the answers.

1. Q&A will be available soon for students who are taking the course remotely.
2. Remote and in-class students can create a project group together.

The academic part of the lecture is summarized in the upcoming sections.

1.2 Why should we learn about distributed systems?

Distributed systems are very common today and designing a distributed system is more complicated than designing standalone system. Most of the online applications that we use on a daily basis are distributed in some shape or form. The examples include World Wide Web (WWW), Google, Amazon, P2P file sharing systems, volunteer computing, grid and cluster computing, cloud computing, etc. It is useful to understand how these real-world systems work.

1.3 What is a distributed system? What are the advantages & disadvantages?

Definition: Two definitions can be given.

1. It is one that has multiple connected CPUs.
2. A collection of multiple machines that appears to its users as a single coherent system.

The examples include parallel machines and networked machines. Distributed systems have the following advantages:

1. **Resource sharing.** Distributed systems enable communication over the network and resource sharing across machines (e.g. a process on one machine can access files stored on a different machine).
2. **Economic.** Distributed systems lead to better economics in terms of price and performance. It is usually more cost effective to buy multiple inexpensive small machines and share the resources across those machines than buying a single large machine.
3. **Reliability.** Distributed systems have better reliability compared to centralized systems. When one machine in a distributed system fails, there are other machines to take over its task and the whole system can still function. It is also possible to achieve better reliability with a distributed system by replicating data on multiple machines.
4. **Scalability.** As the number of machines in a distributed system increases, all of the resources on those machines can be utilized which leads to performance scaling up. However, it is usually hard to achieve linear scalability due to various bottlenecks (more in Section 1.6).
5. **Incremental growth.** If an application becomes more popular and more users use the application, more machines can be added to its cluster to grow its capacity on demand. This is an important reason why the cloud computing paradigm is so popular today.

Distributed systems also have several disadvantages:

1. **High complexity.** Distributed applications are more complex in nature than centralized applications. They also require distribution-aware programming languages (PLs) and operating systems (OSs) which are more complex to design and implement.
2. **Network connectivity essential.** Network connectivity becomes essential. If the connection between components breaks a distributed system may stop working.
3. **Security and Privacy.** In a distributed system, the components and data are available over the network to legitimate users as well as malicious users trying to get access. This characteristic makes security and privacy more serious problems in distributed systems.

1.4 Transparency in Distributed Systems

A transparency is hiding some details from the user. When you build a distributed system, you do not want to expose everything to the user. A general design principle is that if an aspect of the system can be made transparent to its users, then it should be because that would make the system more usable. For example, when a user searches with Google they would only interact with the search box and the results web page. The fact that the search is actually processed on hundreds of thousands machines is hidden from the user (replication transparency). If one of the underlying servers fails, instead of reporting the failure to the user or never returning a result, Google will automatically handle the failure by re-transmitting the task to a back-up server (failure transparency). Although incorporating all the transparency features reduces complexity for users, it also adds complexity for the system. Overall, as a good way of designing systems, the things that users do not need to worry about should be hidden and the things that users need to know should be exposed.

Here are some transparencies:

- **Location.** It hides the location of a resource from the user. For example, when type the URL of facebook.com, we do not know where machines that serve our request are located.

- **Replication.** Replication information of the service is hidden from the user.
- **Failure.** Some elements of your system are hidden from the user. If something goes down, requests are sent to other running nodes.

Question (previous year): Won't the maintenance of several machines out-weight the cost of one better computer (supercomputer)?

Answer: It depends. In general, it is cheaper to buy several machines to get the performance required. On the other hand, this means more hardware that can break or need maintenance. In general we see that several machines are often cheaper than a supercomputer.

Question (previous year): Are there scenarios where you actually ought to reveal some of these features rather than making them transparent?

Answer: There are many systems where you may not want to make something transparent. An example is that if you want to ssh to a specific machine in a cluster, the fact that there is a cluster of machines is not hidden from the user because you want the user to be able to log into a specific machine. So there are many scenarios where having more than one server does not mean you want to hide all the details. The system designer needs to decide what to hide and what to show in order to let the user accomplish their work.

Question (previous year): What does a *resource* mean?

Answer: The term resource is used broadly. It could mean a machine, a file, a URL, or any other object you are accessing in the system.

1.5 Open Distributed Systems

Open distributed systems are a class of distributed systems that offer services with their APIs openly available and published. For example, Google Maps has a set of published APIs. You can write your own client that talks with the Google Maps server through those APIs. This is usually a good design choice because it enables other developers to use the system in interesting ways that even the system designer could not anticipate. This will bring many benefits including interoperability, portability, and extensibility.

1.6 Scalability Problems and Techniques

It is often hard to distribute everything you have in the system. There are three common types of bottleneck that prevent the system from scaling up:

- **Centralized services.** This simply means that the whole application is centralized, i.e., the application runs on a single server. In this case, the processing capacity of the server will become a bottleneck. The solution is to replicate the service on multiple machines but this will also make the system design more complicated.
- **Centralized data.** This means that the code may be distributed, but the data are stored in one centralized place (e.g. one file or one database). In this case access to the data will become a bottleneck. Caching frequently-used data or replicating data at multiple locations may solve the bottleneck but new problems will emerge such as data consistency.
- **Centralized algorithms.** This means that the algorithms used in the code make centralized assumptions (e.g. doing routing based on complete information).

The following are four general principles for designing good distributed systems:

1. **No machine has a complete state information.** In other words, no machine should know what happens on all machines at all times. Here, state can be thought as data or file.
2. **Algorithms should make decisions based on a local information as opposed to global information.** When you want to make a decision, you do not want to ask every machine what they know. For example, as much as possible, when request comes in, you want to serve this using your local information as opposed to coordinating with lots of other machines. The more coordination is needed, the worse your scalability is going to be.
3. **Failure of any one component does not bring down the entire system.** One part of an algorithm failing or one machine failing should not fail the whole application/system. This is hard to achieve.
4. **No assumptions are made about a global clock.** A global clock is useful in many situations (e.g. in a incremental build system) but you should not assume it is perfectly synchronized across all machines.

There are some other techniques to improve scalability such as asynchronous communication, distribution, caching, and replication.

Question (previous year): What is an example of making decisions based on local and global information?

Answer: We will talk about distributed scheduling in a later lecture. As an example, suppose a job comes in to a machine and the machine gets overloaded. The machine wants to off-load some task to another machine. If the machine can decide which task can be off-loaded and which other machine can take the task without having to go and ask all of the other machine about global knowledge, this is a much more scalable algorithm. A simple algorithm can be a random algorithm where the machine randomly pick a machine and says “Hey, take this task, I’m overloaded.” That is making the decision locally without finding any other information elsewhere.

Question (previous year): If you make decisions based on local information, does that mean you may end up using inconsistent data?

Answer: No. The first interpretation of this concept is that everything the decision needs is available locally. When I make a decision I don’t need to query some other machines to get the needed information. The second interpretation is that I don’t need *global knowledge* in order to make a local decision.

1.7 Distributed Systems History and OS Models

Minicomputer model: In this model, each user has its local machine. The machines are interconnected, but the connection may be transient (e.g., dialing over a telephone network). All the processing is done locally but you can fetch remote data like files or databases.

Workstation model: In this model, you have local area networks (LANs) that provide a connection nearly all of the time. An example of this model is the Sprite operating system. You can submit a job to your local workstation. If your workstation is busy, Sprite will automatically transmit the job to another idle workstation to execute the job and return the results. This is an early example of resource sharing where processing power on idle machines are shared.

Client-server model: This model evolved from the workstation model. In this model there are powerful workstations who serve as dedicated servers while the clients are less powerful and rely on the servers to do their jobs.

Processor pool model: In this model the clients become even less powerful (thin clients). The server is a pool of interconnected processors. The thin clients basically rely on the server by sending almost all their tasks to the server.

Cluster computing systems / Data centers: In this model the server is a cluster of servers connected over high-speed LAN.

Grid computing systems: This model is similar to cluster computing systems except for that the server are now distributed in location and are connected over wide area network (WAN) instead of LAN.

WAN-based clusters / distributed data centers: Similar to grid computing systems but now it is clusters/data centers rather than individual servers that are interconnected over WAN.

Virtualization

Cloud computing: Infrastructures are managed by cloud providers. Users only lease resources on demand and are billed on a pay-as-you-go model.

Emerging Models - Distributed Pervasive Systems: The nodes in this model are no longer traditional computers but smaller nodes with microcontroller and networking capabilities. They are very resource constrained and present their own design challenges. For example, today's car can be viewed as a distributed system as it consists of many sensors and they communicate over LAN. Other examples include home networks, mobile computing, personal area networks, etc.

1.8 Operating Systems History

1.8.1 Uniprocessor Operating Systems

Generally speaking, the roles of operating systems are (1) resource management (CPU, memory, I/O devices) and (2) to provide a virtual interface that is easier to use than hardware to end users and other applications. For example, when saving a file we do not need to know what block on the hard drive we want to save the file. The operating system will take care of where to store it. In other words, we do not need to know the low-level complexity.

Uniprocessor operating systems are operating systems that manage computers with only one processor/core. The structure of uniprocessor operating systems include

1. **Monolithic model.** In this model one large kernel is used to handle everything. The examples of this model include MS-DOS and early UNIX.

2. **Layered design.** In this model the functionality is decomposed into N layers. Each layer can only interact with with layer that is directly above/below it.

3. **Microkernel architecture.** In this model the kernel is very small and only provides very basic services: inter-process communication and security. All other additional functionalities are implemented as standard processes in user-space.

Question (by Instructor): What is the drawback of microkernel architecture?

Answer (Student): Communication between OS modules becomes bottleneck.

Answer (Instructor cont.): There is a lot of communication that has to happen between all modules for an OS to achieve its task. For example, when you start a new process, you have to send a message to the memory module to request RAM. All of this message passing actually kills the performance quite a bit.

Hybrid architecture: Some functionalities are independent processes while other functionalities are moved back to kernel for better performance.

1.8.2 Distributed Operating System

Distributed operating systems are operating systems that manage resources in a distributed system. However, from a user perspective a distributed OS will look no different from a centralized OS because all of the details about distribution are automatically handled by the OS and are transparent to the user.

There are essentially three flavors of distributed OS's: distributed operating system (DOS), networked operating system (NOS), and middleware. DOS provides the highest level of transparency and the tightest form of integration. In a distributed system managed by DOS, everything operates above the DOS kernel will see the system as a single logical machine. In NOS you are still allowed to manage loosely-coupled multiple machines but it does not necessarily hide anything from user. Middleware takes a NOS and adds a software layer on top to make it behave or look like a DOS.

CMPSCI 677 Distributed and Operating Systems **Spring 2022**

Lecture 2: January 31

Lecturer: ***Prashant Shenoy*** *Scribe:* ***Steven (Jiaxun) Tang (2022), Roy Chan (2019), Phuthipong Bovornkeeratiroj (2018)***

Note: Please make sure that Gradescope is working for you.

Note: There is a Career Fair on February 24.

Reminder: No using laptops or phones during class.

2.1 Architectural Styles

Most distributed systems can be described by one of the architectures discussed in this lecture. It is important to understand the differences between them so that we can decide on the architecture before implementing a new system.

2.1.1 Layered Architectures

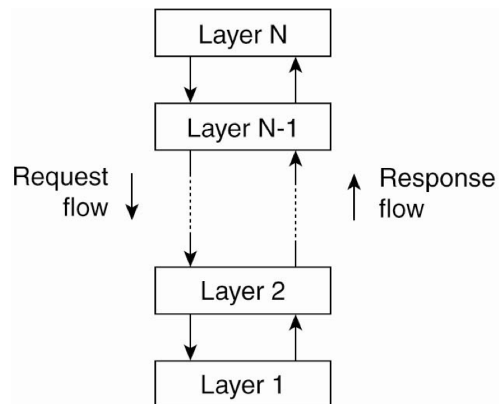


Figure 2.2: Layered Design

A layered architecture looks like a stack, as seen in the figure above. The system is partitioned into a sequence of layers and each layer can communicate to the layer above or below. For example, layer i can communicate with layer $i+1$ and layer $i-1$ but not the others (e.g. layer $i+2$). This is the main restriction of a layered design. The layered architecture is especially common in web applications where this architecture is divided across the client and the server. Common instances of these systems are multitiered architectures and network stack.

2.1.2 Object-Based Style

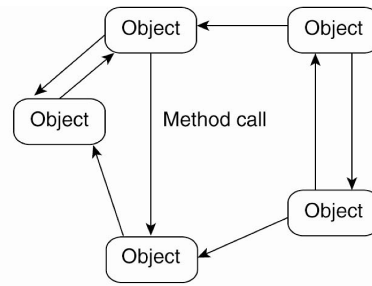


Figure 2.3: Object-based Style

In this architecture, each component corresponds to an object. Unlike in standard OOP programming, objects can be distributed across multiple machines. As shown in the figure above, the system can have many objects. Each object have has its own states and exposes its own interface which other objects can use. All objects can communicate with any other object without restriction, making this a "generalized" version of the layered design. Components interact with each other via remote procedure calls. We will discuss RPC in Lecture 3.

2.1.3 Event-Based Architecture

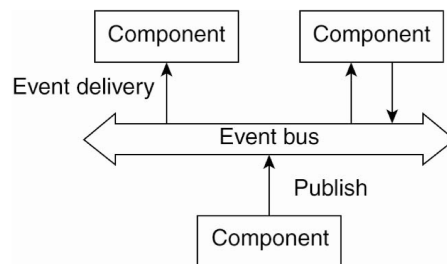


Figure 2.4: Event-based Architecture

An event-based architecture has many components that communicate using a publisher-subscriber (pub-sub) model via an event bus instead of through direct communication. In this architecture, a component that sends an event to the event bus is a publisher, and a component that subscribes to certain types of events on the event bus is a subscriber. Each component will work asynchronously. After a component sends information by publishing an event, the event bus then checks for subscriptions matching the recipient information enclosed in the newly published event. If one or more matching subscriptions is found, the event bus will deliver the data to the appropriate component(s). There are many kinds of event busesk, e.g., memory-based or disk-based.

2.1.4 Shared Data Space

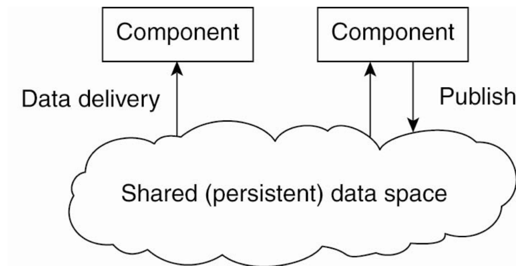


Figure 2.5: Shared Style

The shared data space architecture has a shared data space which is like a physical bulletin board. A component posts information and some component may come along later and retrieve the information. Unlike in the event-based architecture, data posted in the shared data space have no specific information about the recipient. Therefore, posted data can be in the shared data space for a while until some component actively retrieve this data. From this sense, the components in the data-space architecture are loosely coupled in space and time. Notice that the data that is published is not addressed to anyone in particular, and that the data may not be recieved in real-time.

Question (Student): Does the shared data space architecture require a centralized server?

Answer (Instructor): The shared data space model is like a distributed database. It does not have to be centralized and can be done in a distributed manner.

2.1.5 Resource-Oriented Architecture (ROA)

A resource-oriented architecture exposes resources for clients to interact with. Resources have names and related operations. Representational State Transfer (REST) is a common implementation of this architecture. It has a standard naming scheme in which all services offer the same interface (GET/PUT/POST/DELETE). No client state is kept, which means each request is logically decoupled. Since users often interact with the resources of a web service, exposing application as resources make it easy to implement descriptive APIs. For example, if you want to query/create/delete/update an object in the Amazon Object Storage Service S3, you just need to send GET/PUT/DELETE/POST requests to `https://{{BUCKET_NAME}}.s3.aws.com/{{OBJECT_NAME}}`.

Question (Student): REST services are stateless because all the information is in the URL. What about authentication? Would you maintain states for that?

Answer (Instructor): It all depends on how you design the web service. Once authenticated, it can provide a token. You can do authentication in a stateless manner.

Question (Student): Wouldn't there still be a state sometimes?

Answer (Instructor): The service will typically be stateless.

2.1.6 Service-Oriented Architecture

A service-oriented architecture exposes components as services. Each component provides a service. Services communicate with each other to implement an application. Micro-services are one modern implementation

of a service-oriented architecture.

Question (Student): What is the difference between SOA and ROA?

Answer (Instructor): SOA exposes components as services and ROA exposes components as resources. ROA requires services to connect via HTTP, while SOA doesn't enforce the protocol used. ROA is stateless and SOA can be stateful. ROA is also newer and is better for using HTTP.

Answer (Instructor): ROA is stateless and SOA can be stateful. ROA is also newer and is better for using HTTP.

The following are comparisons between OOA, ROA, and SOA.

Attribute	Object-oriented	Resource-oriented	Service-oriented
Granularity	Object instances	Resource instances	Service instances
Main Focus	Marshalling parameter values	Request addressing (usually URLs)	Creation of request payloads
Addressing / Request routing	Routed to unique object instance	Unique address per resource	One endpoint address per service
Are replies cacheable?	No	Yes	No
Application interface	Specific to this object / class – description is middleware specific (e.g. IDL)	Generic to the request mechanism (e.g. HTTP verbs)	Specific to this service – description is protocol specific (e.g. WSDL)
Payload / data format description	Yes – usually middleware specific (e.g. IDL)	No – nothing directly linked to address / URL	Yes – part of service description (e.g. XML Schema in WSDL)

Courtesy: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/3-arch-styles.pdf

Figure 2.6: OOA vs. ROA vs. SOA

Question (Student): Can a system use hybrid architectures?

Answer (Instructor): Yes. For example, an application that uses micro-services can also implement RESTful API. You can often choose a base architecture and compose other architectures together according to your needs.

2.2 Client-Server Architecture

This is the most popular architecture. The client sends requests to the server, and then the server sends a response back to the client. Remember that this does not necessarily refer to the hardware. The terms "client" and "server" refer instead to the piece of software that requests the service or provides the service. After the client sends a request, it waits while the server processes the request. In the figure below, you can see the respective parties waiting when there is a dotted line.

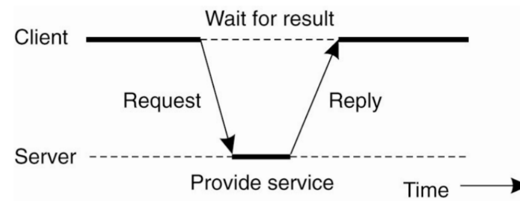


Figure 2.6: Client-Server Architecture

Developers need to make design choices about which service should be put into which layer. Let us look at an example to see how we would implement this.

2.2.1 Search Engine Example

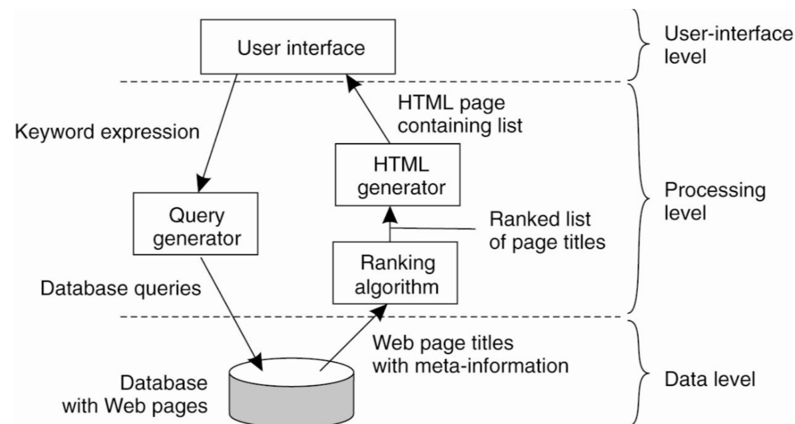


Figure 2.7: Search Engine Example

Take Google search as an example. When you type something into the search box, you are interacting with the UI level of Google search. Then, the UI will send your input to a query generator at the processing level. The query generator translates your query expression into database queries and accesses the database located at the data level. A ranking algorithm in the processing level takes the query results, ranks them, and passes the result to the HTML generator at the same level. The generated page is then sent back to the UI layer and will be rendered by the browser as a webpage. The important part is understanding the tiers and how they interact with each other in a distributed application. Other details like indexes and crawlers are not the components we are considering here.

2.2.2 Multitiered Architectures

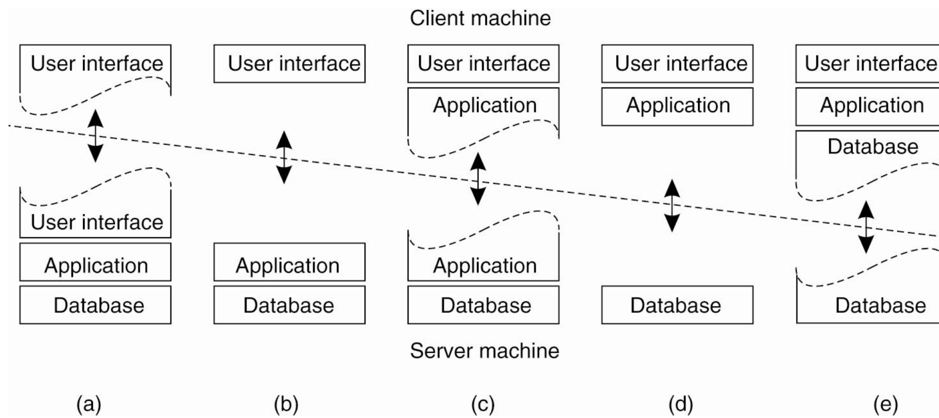


Figure 2.8: Client Server Choices

We see various "splits" of the 3 layers between client and server represented by the dotted line. The layer(s) above the line are on the client, and the layer(s) below the line are on the server. As you can see, there are many choices in how you split the implementation.

A typical implementation of (a) is a traditional browser-based application (e.g. SPIRE). The webpage is constructed from the server-side and rendered in the browser. A typical implementation of (b) is single-paged web application. The server does not render pages, but only provides APIs for data retrieval. The browser will send AJAX requests to call those APIs. A typical implementation of (c) is a smartphone app, where the application's backend is usually split between the device and the server. Desktop applications usually follow (d) where only the database is on the server, and the client is just accessing data. A smartphone app or a whole app that exists on a client also follows this architecture. Lastly, (e) improves on (d). Data is cached or stored locally. For example, Google's offline mail caches a small subset of the user's email locally. The choice of which architecture to use depends on many factors, e.g., what you want to do, how much resources the client has, etc.

2.2.3 Three-tier Web Application

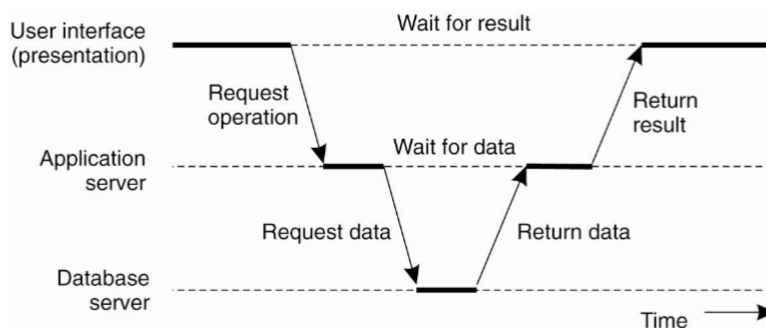


Figure 2.9: Three-tier Web Application

The three-tier web application architecture is a very popular architecture choice. The client's browser sends an HTTP request to an HTTP server (e.g. apache). The HTTP server then sends the request to the app

server (e.g. a Python backend) for processing in which it may create a query to the database server. The database returns data to the app server that sends the results to the HTTP server which then forwards it to the browser. The sequential nature of this architecture is a type of layer architecture seen earlier in the search engine example.

These tiered architectures can use more or fewer than 3 layers depending on their setup. Modern web applications will take the Application tier and split it into multiple tiers. A very common architecture for web apps uses HTTP for the user, PHP or J2EE for the app server, and then a database for the bottom tier. The divide between user and server is not set in stone as we saw in the previous section.

2.2.4 Edge-Server Systems

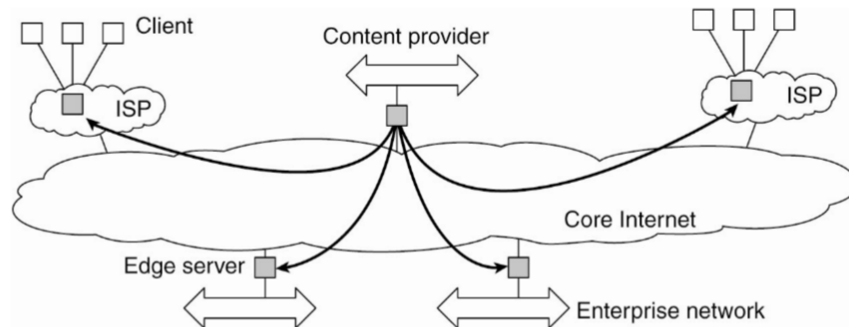


Figure 2.10: Edge Server

Unlike traditional client-server architecture, edge server systems implement a client-proxy-server architecture. As the name suggests, there is an extra component in between. The proxy (labeled as the edge server in the figure above) sees if it can process the client's request without having to go to the server (i.e. the Content provider). If not, the proxy forwards the request to the real server. The advantage of this approach is that the main server load is reduced, and data is moved to servers closer to the user so that the access latency will be greatly reduced. Many other proxy services can be provided in addition to caching. Edge computing goes one step further than simply providing a data cacheby allowing code execution in the edge server.

2.3 Decentralized Architectures (Module 3)

Decentralized architectures are also known as peer-to-peer (P2P) systems. Unlike the client-server architecture, each node (peer) can be a client, server, or both with all nodes being mostly equal. That is, we are removing the distinction between client and server. P2P systems can also come be structured or unstructured systems. A peer can provide services and request services. Peers can also come and go at any time, unlike a server which must be there all the time. We will introduce a structured peer-to-peer system named "Chord" as an example.

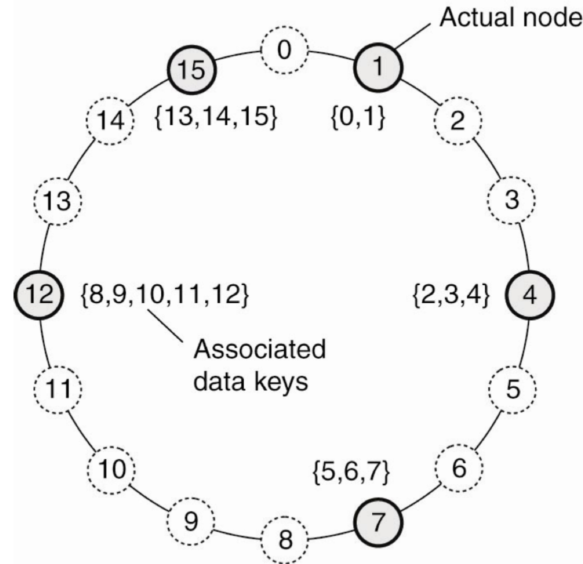


Figure 2.11: Chord Structure

The Chord system maintains a hash function to associate data nodes with an integer key. In this figure, there are $n = 16$ keys in the system. The darker circles are peers that already joined the Chord. Node 1 is responsible for storing data $\{0, 1\}$, node 4 is responsible for storing node $\{2, 3, 4\}$ When a node joins, it picks an ID that is a key and is unfilled from 1 to n and then stores keys from the previous node to itself. How one chooses the key for a joining node can be random or structured. In our current case, when n_7 joined, it became responsible for storing [7, 6, 5]. When a node leaves, the chord structure assigns the leaving node's keys to the next node above it. If n_7 were to leave, n_{12} would then be responsible for $\{12, 11, 10, 9, 8, 7, 6, 5\}$. As one can see, joins and leaves are symmetric. Replication or redundancy is used so that when the node leaves, the system still works.

Given a key in a request, the system has to figure out what node has that key. This can cause request routing, in which the system will hop around nodes until the key has been found. Fortunately, the search is actually fast, with a provided key, the system has to look up the value in the distributed hash table. The hash table is provided by the distributed hash table (DHT) algorithm. P2P architectures are not as reliable as client-server architectures, as peers can join and leave the network without advance notice. A technique called "consistent hashing" ensures the DHT is fault tolerant.

More details about Chord can be found here:
`https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`

Question (Student): What is the value that is stored in the hash table?

Answer (Instructor): It is a hash table so anything that can be in it. Usually it can store a file, i.e., the service is a file lookup. But it is like asking, "What can you store in a database?" Whatever you want!

Question (Student): Does a P2P architecture imply an ad-hoc network? That is, do nodes just come and go?

Answer (Instructor): It is designed to ensure high reliability. Unlike in a client-server architecture where the server is assumed to be reliable, nodes may not be reliable so the system is built to handle nodes joining and leaving.

2.3.1 Content Addressable Network (CAN)

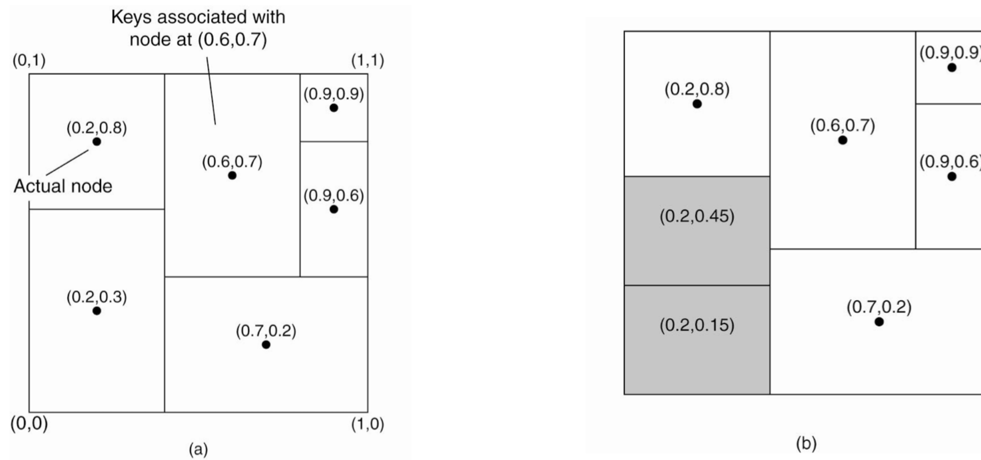


Figure 2.12: CAN Structure, with (b) showing a join procedure

Content Addressable Network (CAN) is another P2P system. As opposed to Chord, however, CANs are generalized versions of Chord, i.e., they use a d-dimensional coordinate system. To make illustrations easier, we will set $d = 2$ for the rest of this section. For example, we can have a tuple containing a file name and a file type which would require a two-part key for the two-part attribute. Here, each piece of content in a CAN has 2 identities: $< id.x, id.y >$ or <file name, file type>. For example, two files named "Foo" may have different file types such as .jpg and .txt.

In the figure above, each dot is a node, meaning that each node is responsible for a rectangular partition of the coordinate space. The user can have a more fine-grained query in this structure. The x-axis and y-axis are showing normalized values of the keys from 0 to 1. If a node joins, it chooses a random (x, y) coordinate and splits the box (i.e. a specific coordinate space) that it is in with the existing node. A node leaving is more difficult, as the merging of 2 rectangles is not always a rectangle. If a node leaves, the system must partition that rectangle to merge it with other already present rectangles. Consistent hashing again ensures the correct handling of the hash when nodes exit.

Note: Remember that the specific example here shows 2 dimensions, but CAN could have any d-dimensional coordinate system.

Note: In Chord, one can also represent the <file name, file type> attribute, but this would require concatenating the 2 keys into one.

Question (Student): Does this require a full replication of content on all peers?

Answer (Instructor): No, because that assumes all nodes will leave at the same time. Say, for example, each node replicated at 3 other nodes. This provides the assumption that if 3 nodes leave, we're still ok.

2.3.2 Unstructured P2P Systems

Rather than adhering to some topological protocol such as a ring or a tree, unstructured topologies are defined by randomized algorithms, i.e., the network topology grows organically and arbitrarily. Each node picks a random ID and then picks a random set of nodes to be neighbors with. The number of nodes is based on the choice of degree. If $k = 2$, it means the new node will randomly link to 2 existed nodes and

establish logical connections. When a node leaves, the connections are severed and any remaining nodes can establish new links to offset the lost connections.

Without structure, certain systems can become more complicated. For example, a hash table key lookup may require a brute force search. This floods the network, and the response also has to go back the way it came through the network. We observe that the choice of degree impacts network dynamics (overhead of broadcast, etc.). The unstructured notion of such P2P systems framed early systems, but newer systems have more structure in order to reduce overhead.

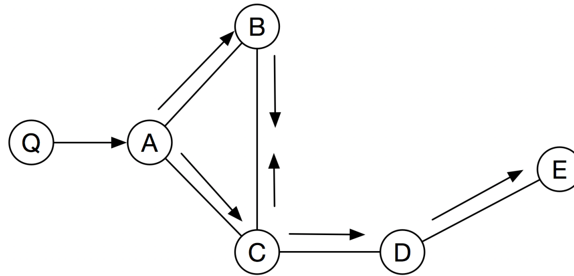


Figure 2.13: Search in Unstructured P2P System

From the figure above, we see that search in an unstructured P2P system is done by propagating through the graph as seen in the above figure. Here, a query (Q) is passed to node A, which is then propagated through the network as each node queries its neighbors. Eventually, the signal is backpropagated to the sender. This can easily flood the system as mentioned above, so one can create a hop count limit to reduce unnecessary traffic. Each time the query is passed to a neighbor, the hop count is decremented. Upon reaching 0, the node will simply return not found.

2.3.3 SuperPeers

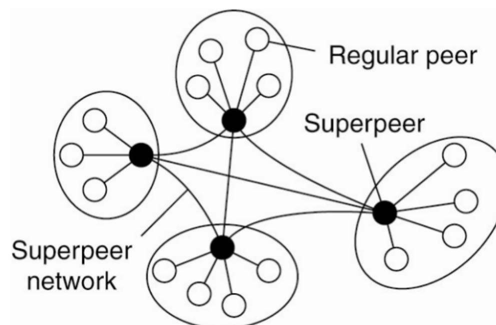


Figure 2.15: Graph with SuperPeer Structure

A small modification to the completely unstructured P2P system allows for much more efficient communication and reduces overhead. The P2P graph is partitioned into clusters, where one peer, designated to be the superpeer, within each cluster can communicate with other peers outside of the cluster. These superpeers are dynamically elected within each group and should have additional resources to facilitate the increased communication demand.

The restricted communication reduces unneeded calls to neighbors and prevents the huge amount of broadcast

traffic found in the completely unstructured P2P system. The number of messages should be lower. However, there may still be a lot of traffic still flooding the network albeit only going through superpeers.

An early versions of Skype was a good example of how superpeers work. It tracked where users were and if they were logged in from a specific cluster. It was a P2P system, but Skype has now moved to a client-server architecture instead.

Question (Student): What are some more examples of superpeers?

Answer (Instructor): BitTorrent and P2P backup systems. However, whenever an application is very important, they may not use P2P since P2P assumes that people are donating resources to make the system work.

Question (Student): Are node link connections static or dynamic?

Answer (Instructor): We can't assume that neighbors will stay up. The topology is constantly changing so we must assume dynamic connections and that links with new neighbors will be made.

2.3.4 Collaborative Distributed Systems

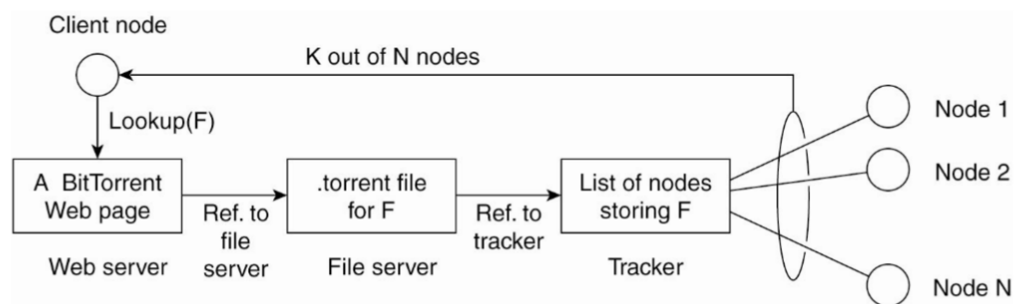


Figure 2.16: BitTorrent, an example of a collaborative distributed system

In a collaborative distributed system, files are split into chunks and spread across peers. A client can request these chunks and piece them together. This system allows parallel file download sources from multiple connections, which is faster than a sequential file download from a singular connection. A node can control how parallel it wishes to be, i.e., how many nodes or peers it connects to.

A system like BitTorrent can also take into account an altruism ratio, and slow down a node based on the ratio. If a node is just downloading without also uploading chunks in its possession, or more generally, provide services to other peers, then the system may reduce the download speed of the node. This incentivizes nodes to participate in and contribute to the network instead of freeloading so that they can get good performance.

Two key components are involved in a torrent system: the tracker and the torrent file. The tracker is an index that monitors which nodes have which chunks. The torrent file points to the tracker and can be posted on a web server. In short, the torrent file gets a client node to the tracker which shows which peers it needs, and then the client node can directly connect to those peers based on the configurable setting of how many peers it wants to connect to at one time.

Question (Student): Does the tracker get updated?

Answer (Instructor): As long as a user is connected to it, the tracker knows who has what content.

Question (Student): How do nodes agree on how a file is split?

Answer (Instructor): The file is split how you want. This is a configurable parameter in the system.

2.3.5 Autonomic Distributed Systems

An autonomic distributed system can monitor itself and take action autonomously when needed. Such systems can perform actions based on the system performance metrics, system health metrics, etc. We will not dive too deep into this topic, but knowing the concept helps.

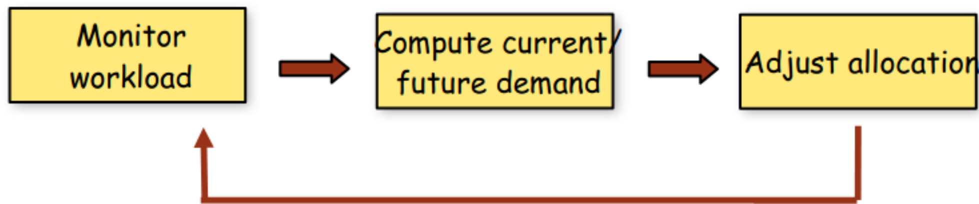


Figure 2.17: Automatic capacity provisioning

This is an illustration of how you might implement such a system. You can monitor the current workload, predict future demand, and if the system thinks the current resource is not enough, deploy more nodes. If the system thinks the resource is not enough, then the number of nodes could be reduced. This technique is also called elastic scaling. The workload prediction part can involve many techniques. For example, we could use feedback and control theory to design a self-managing controller. Machine learning techniques such as reinforcement learning can also be used.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 3: February 2

Professor: Prashant Shenoy *Scribe: Rajul Jain (2022)*

3.1 Message-Oriented Communication

3.1.1 Communication between processes

Suppose two processes running on same machine want to communicate with eachother. There are 2 ways to achieve this.

Unstructured Communication:

- Processes use a shared memory/buffer or a shared data structure for communication.
- This type of communication is called "unstructured" because the buffer is just a piece of memory and has no structure associated with it. In other words, we can put any data that we want in the buffer and the processes have to interpret that data and do something with it.

Structured Communication:

- Processes send explicit messages to communicate, i.e., using Inter-Process Communication (IPC).
- This can be achieved by either low-level socket-based message passing or higher-level remote procedure calls.

Now suppose the two processes are running on different machines in a distributed system. If the processes communicate via unstructured communication, then we need to figure out a way to share buffers across machines. Clearly, making memory accessible across machines connected over a network is more difficult than using shared memory in the same machine. Hence, we prefer structured communication like sockets and RPCs.

Question: Why do distributed systems need low-level communication support for both structured and unstructured communication?

Answer: Because distributed systems are spread over a network.

3.1.2 Communication (Network) Protocols

Suppose two processes (App 1 and App 2), each using the OSI or TCP/IP model, communicate over a network. App 1 creates a message at the application layer which travels down the network stack layers (where each layer adds their headers to the original message) all the way to the physical layer. The message is then passed over to App 2's physical layer where it travels all the way up the network stack again to complete the communication.

3.1.3 Middleware Protocols

In a distributed system, *middleware* is the layer residing between the OS and an application. The middleware layer sits between the application and transport layer in the network protocol stack.

3.1.4 TCP-based Socket Communication

TCP-based socket communication is a structured communication method which uses TCP/IP protocols to send messages. The socket creates network address (IP address and port number) for communication. The socket interface creates network end-points. For example, in a client-server distributed system, both the client and the server need to create sockets which are bound to port number where they can listen to and send messages.

3.1.4.1 Understanding TCP Network Overheads

Consider a client-server model using TCP method for communication. As TCP method follows a 3-way handshake protocol for communication, this leads to a transfer of 9 messages for sending a single request and subsequently receiving a single response. This is a huge overhead.

3.1.5 Group Communication

In a distributed system, when one machine needs to communicate with many machines, group communication protocols are used. This is analogous to sending an email to multiple recipients. For group communication, all the recipients subscribe to a *group address*. The network then takes care of delivering the messages to all the machines in the group address. This is called *multicasting*. If the messages are sent to all the machines in the network, then the process is called *broadcasting*.

3.2 Remote Procedure Calls

RPCs provide higher level abstractions by making distributed computing look like centralized computing. They automatically determine how to pass messages at the socket level without requiring the programmer to directly implement the socket code. In other words, instead of sending message to the server to invoke a method X, the programmer can call the method X directly from the client machine. RPCs are built using sockets underneath. The programmers do not write the this socket code. Instead, it is auto-generated by the RPC compiler. Stubs are another piece of RPC compiler auto-generated code which convert parameters passed between client and server during RPC calls.

There are a few semantics to keep in mind:

- Calling a method using RPC is same as invoking a local procedure call.
- One difference is that in an RPC, the process has to wait for the network communication to return before it can continue its execution, i.e., RPCs are *synchronous*.
- *You cannot pass pointers or references.* Pointers and references point to a memory location in the respective machine. If these memory locations are passed on a different machine, they will point to something completely different on that machine.

- *You cannot pass global variables.* As global variables lying on one machine cannot be accessed by another machine simply over a network. Hence global variables are not allowed in an RPC.
- *Pass arguments by value.*

Question: If we absolutely need to pass pointers in an RPC, how do we achieve that?

Answer: Take the entire object (say, from the client machine) and pass it on to the server. Create a copy of this object on server and then create a local pointer to the object.

3.2.1 Marshalling and Unmarshalling

Different machines use different representation of data formats. This creates discrepancy in understanding messages and data between different machines. Hence before sending messages to the other machine, the messages are converted into a standard representation such as eXternal Data Representation (XDR). Marshalling is the process of converting data on one machine into a standard representation suitable for storage/transmission. Unmarshalling is the process of converting from a standard representation to an internal data structure understandable by the respective machine.

3.2.2 Binding

The client locates the server using bindings. The server that provides the RPC service registers with a naming/directory service and provides all of the details such as method names, version number, unique identifier, etc. When client needs to access a specific method/functionality, it will search in the naming/directory service if there's a service in the network which hosts this method or not and then accesses the server using the IP and port number listed in the directory.

3.3 RPC Implementation

3.3.1 Failure semantics

- Lost request/ response messages: The network protocols handle these errors by timeout mechanisms.
- Server and client failures: Need to be handled while creating distributed systems.

If client crashes after sending a request to the server, then the server response is referred to as **orphan**. To deal with the orphans, methods such as Extermination, Reincarnation, Gentle reincarnation, and expiration can be used.

3.3.2 Case study: SUNRPC

SUNRPC was developed for use with NFS. It is one of the widely used RPC systems. Initially, it was built on top of UDP, but later transferred to TCP. It uses SUN's XDR format.

Lecture 4: February 7

Lecturer: Prashant Shenoy *Scribe: Ge Shi (2019), Shubham Shetty (2022)*

4.1 Overview

In the previous lecture, we learned about remote procedure call, socket programming, and communication abstractions for distributed systems. This lecture will cover the following topics:

- Alternate RPC Models
- Remote Method Invocations (RMI)
- RMI & RPC Implementations and Examples

4.2 Alternate RPC Models

4.2.1 Lightweight RPC (LRPC)

Many RPCs occur between client and server on same machine. Lightweight RPCs are the special case of RPCs which are optimised to handle cases where calling process and the called process are on the same machine.

Remember the two forms of communication of a distributed system – explicit (passing data) and implicit (sharing memory). You can think of not using an RPC system for the special case that both processes are on the same machine but using a shared piece memory. The optimization is to construct the message as a buffer and simply write to the shared memory region. This avoids the TCP/IP overheads associated with normal RPC calls.

When client and server both are two processes on the same machine and you make RPC calls between two components on the same machine, following are the things which can make it better over the traditional RPC:

1. No need for the marshalling here.
2. We can get rid of explicit message passing completely. Rather shared memory is used as a way of communication.
3. Stub can use the run-time flag can be used to decide whether to use TCP/IP (Normal RPC) or shared memory (LRPC).
4. No XDR is required.

Steps of execution of LRPC:

1. Arguments of the calling process are pushed on the stack,
2. Trap to kernel is send,
3. After sending trap to kernel, it either constructs an explicit shared memory region and put the arguments there or take the page from stack and simply turn it into shared page,
4. Client thread executed procedure (OS upcall),
5. Then the thread traps to the kernel upon completion of the work,
6. Kernel again changes back the address space and returns control to client,

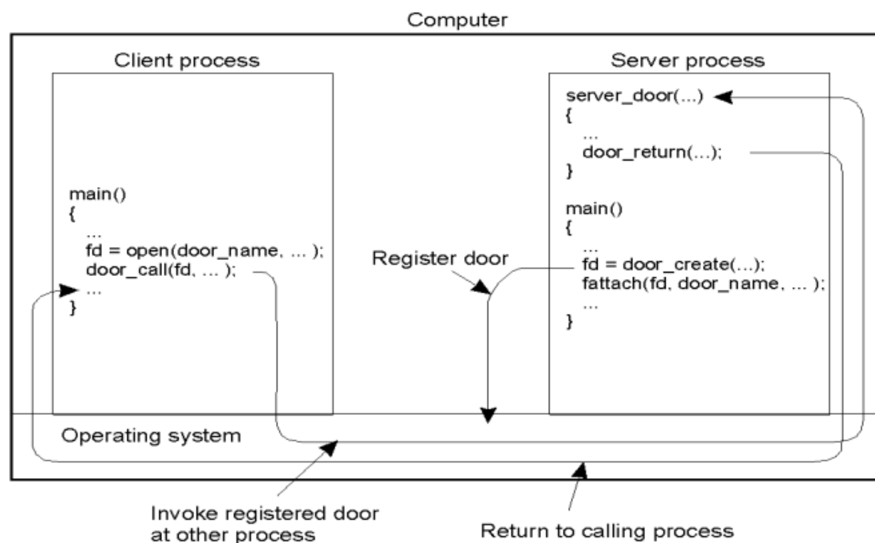


Figure 4.1: Lightweight RPC

Q: If the two processes are on the same machine, why do we need to use a networks stack because you are not communicating over network.

A: If you are using standard RPCs, your package will go down OS and the IP layer will realize they are the process on the same machine. It will come back up and call another process.

Note: RPCs are called "doors" in SUN-OS (Solaris).

4.3 Other RPC models

Traditional RPC uses Synchronous/blocked RPC, where the client gets blocked making an RPC call and gets resumed only after getting result from the called process. There are three other RPC models described below:

4.3.1 Asynchronous RPC

In Asynchronous or non-blocking RPC call, the client is not blocked after making an RPC call. Rather, client sends the request to the server and waits for an acknowledgement from the called process. Server can reply as soon as request is received and execute the procedure later. After getting the acceptance, client resumes the execution.

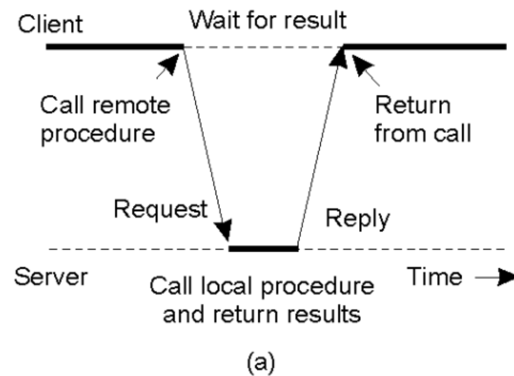


Figure 4.2: Traditional (Synchronous) RPC

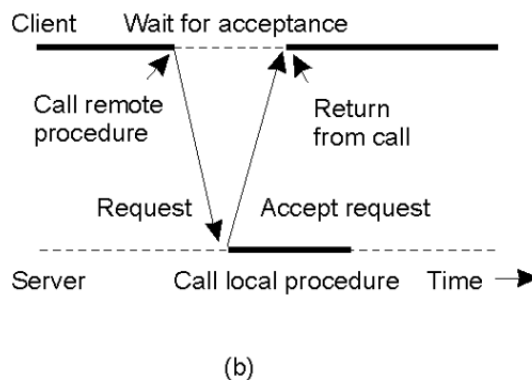


Figure 4.3: Asynchronous RPC

4.3.2 Deferred synchronous RPC

This is just a variant of non-blocking RPC. Client and server interact through two asynchronous RPCs. As compared to Asynchronous RPC, here the client needs a response for server but cannot wait for it, hence the server responds via its own asynchronous RPC call after completing the processing.

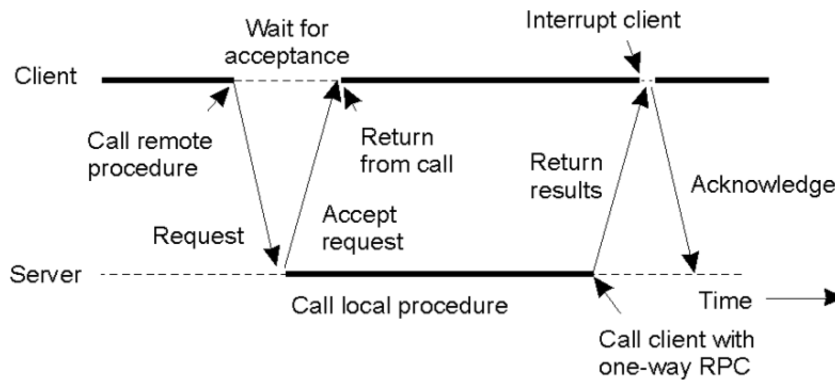


Figure 4.4: Deferred Synchronous RPC

4.3.3 One-way RPC

It is also a form of asynchronous RPC where the client does not even wait for an acknowledgment from the server. Client continues with its own execution after sending RPC call. This model has one disadvantage that it doesn't guarantee the reliability as the client doesn't know whether the request reaches the server or not.

4.4 Remote Method Invocation (RMI)

RMIs are RPCs in Object Oriented Programming Mode i.e., they can call the methods of the objects (instances of a class) which are residing on a remote machine. Here the objects hide the fact that they are remote. The function is called just like it is called on a local machine. For eg: obj.foo() , where *obj* is the object and *foo* is its public function.

Some important facts about RMIs:

1. There is separation between interface and the implementation as the interface is residing on the client machine whereas the implementation is on server machine.

2. It supports system-wide object references i.e parameters can be passed as object references here (which is not possible in normal RPC)

Figure 4.5 is showing an RMI call between the distributed objects. Just like a normal RPC, here also there is no need to setup socket connections separately by the programmer. Client stub is called the *proxy* and the server stub is called the *skeleton* and the instantiated object is one which is grayed in figure.

Now, when the client invokes the remote method, the RMI call comes to the *stub* (Proxy), it realizes that the object is on the remote machine. So it sets up the TCP/IP connection and the marshalled invocation is sent across the network. Then the server unpacks the message, perform the actions and send the marshalled reply back to the client.

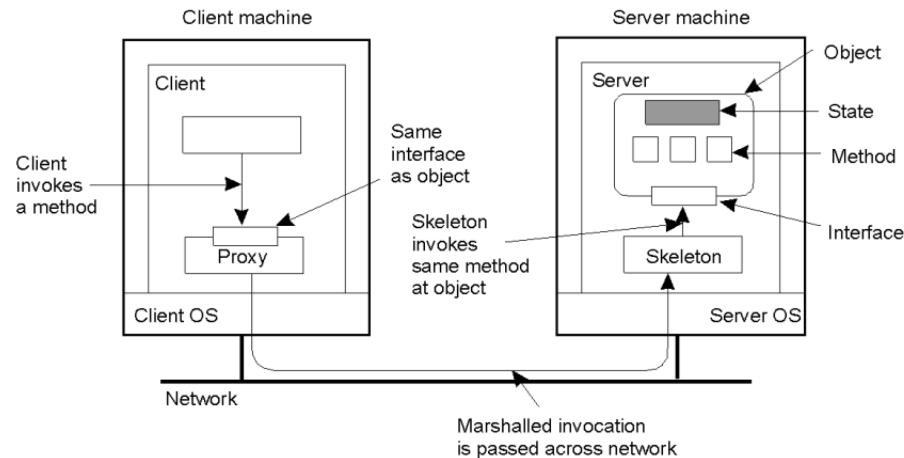


Figure 4.5: Distributed Objects

4.4.1 Proxies and Skeletons : Client and Server Stub

- Working of Proxy : Client stub
  1. Maintains server ID, endpoints, object ID.
  2. Sets up and tears down connection with the server
  3. Serializes (Marshalling) the local object parameters.
- Working of Skeleton : Server stub
  It deserializes and passes parameters to server and sends results back to the proxy.

Q: If the remote object has some local state (variable), you make a remote call and changed the variable, will the local machine see the change?

A: There's only one copy on the server and no copy of it on the client at all. The objects on the server and client are distinct objects. The change will be visible to subsequent methods from client. It doesn't mean there's a copy of the object on the client. If you make another call and see what's the value that variable, you'll get the new one.

4.4.2 Binding a Client to an Object

Binding can be of two types : implicit and explicit. Section (a) of Figure 4.6 shows an implicit binding, which is using just the global references and it is figured out on the run-time that it is a remote call (by the client stub). In section (b), explicit binding is shown, which is using both global and local references. Here, the client is explicitly calling a bind function before invoking the methods. Main difference between both the methods is written in Line 4 of the section (b), where the programmer has written an explicit call to the bind function.

```
Distr_object* obj_ref;            //Declare a systemwide object reference
obj_ref = …;                      // Initialize the reference to a distributed object
obj_ref-> do_something();         // Implicitly bind and invoke a method
                          (a)
Distr_object obj_ref;             //Declare a systemwide object reference
Local_object* obj_ptr;            //Declare a pointer to local objects
obj_ref = …;                      //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);          //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();        //Invoke a method on the local proxy
                          (b)
```

Figure 4.6: Implicit and Explicit Binding of Clients to an Object

4.4.3 Parameter Passing

RMIs are less restrictive than RPCs as it supports system-wide object references. Here, Passing a reference to an object means passing a pointer to its memory address over the network. In Java, local objects are passed by value, and remote objects are passed by reference. Figure 4.7 shows an RMI call from Machine A (client) to the Machine C (server - called function is present on this machine) where Object O1 is passed as a local variable and Object O2 is passed as a reference variable. Machine C will maintain a copy of Object O1 and access Object O2 by dereferencing the pointer.

Note: Since a copy of Object O1 is passed to the Machine C, so if any changes are made to its private variable, then it won't be reflected in the Machine C. Also, Concurrency and synchronization need to be taken care of.

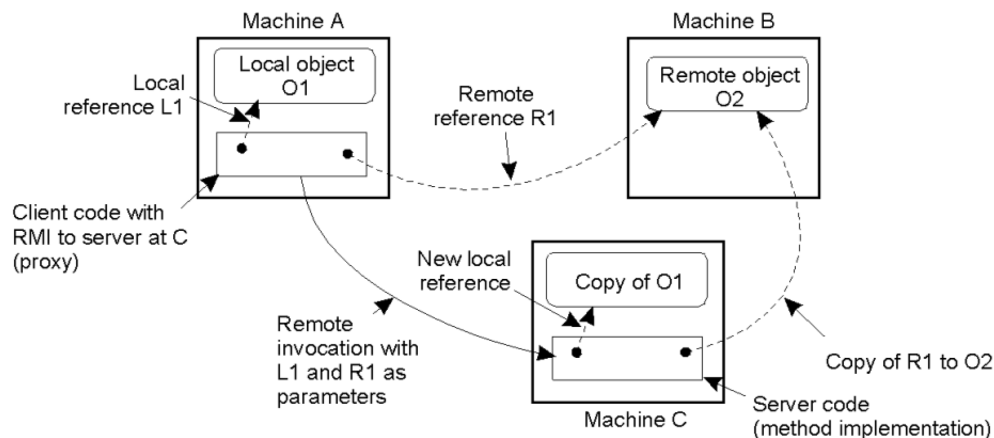


Figure 4.7: Parameter Passing : RMI

Q: How do network pointers interact with Java's garbage collection?
A: Garbage collection of Java is going to delete the memory that is not in use. A short answer is the remote machine shouldn't do garbage collection because you don't know if the object is being used by other machine.

Q: What is a remote reference?
A: A remote reference is an interface which allows to invoke a remote method.

4.5 Java RMI

- Server:
  - The server defines the interface and implements the interface methods. The server program creates a server object and registers object with "remote object" registry (Directory service).
- Client:
  - It looks up the server in remote object registry, and then make the normal call to the remote methods.
- Java tools:
  - `rmiregistry`: Server-side name server
  - `rmic`: Uses server interface to create client and server stub. it is a RMI Compiler, which creates an autogenerated code for stubs.

Figure 4.8: Java RMI Example Code Snippet

Q: How is interface code shared between client and server?
A: Interface is declared in server code and then imported in both client code. Server has to provide an implementation of the interface.

Q: Is Java RMI synchronous or asynchronous?
A: Default abstraction of Java RMI is synchronous.

Q: Where is the RMI registry running?
A: RMi registry can run on any machine. Client and server have to agree on which machine the registry is running on. Default machine is same as server.

4.5.1 Java RMI and Synchronisation

Java supports monitors , which are the synchronised objects. The same method can be used for remote method Invocation which allows concurrent requests to come in and synchronisation them. So for synchronisation, lock has to applied on the object which is distributed amongst the clients. How to implement the notion of the distributed lock?

1. Block at the server : Here, clients will make requests to the server, where they will contend for the lock and will be blocked (waiting for the lock).
2. Block at the client (proxy) collectively : They will have some protocol which decides which client will get the lock and rest others will be blocked (waiting for the lock)

Note: Java uses proxies for blocking (which means client side blocking). Applications need to implement distributed locking.

4.6 C/C++ RPC

Similar to Java RMIs. C++ defines interface in a specification file (*.x* file) which is fed to rpcgen compiler. rpcgen compiler generates stub code, which links server and client C code.

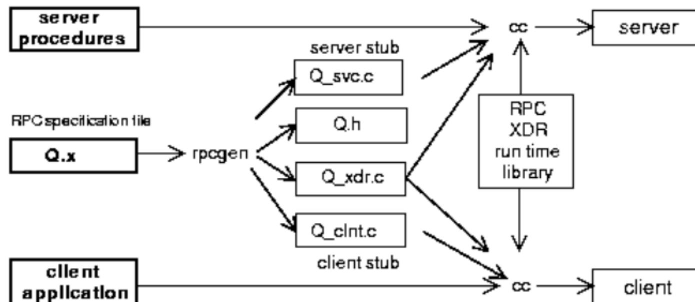


Figure 4.9: rpcgen Compiler

4.6.1 Binder: Port Mapper

Similar to `rmiregistry` in Java, it is a naming server for C/C++. It maintains a list of port mappings. Steps involved for port mapper are -

1. Server start-up: creates a port
2. Server stub calls `svc_register` to register program number, version number with local port mapper.
3. Port mapper stores prog number, version number, and port
4. Client start-up: call `clnt_create` to locate server port
5. Upon return, client can call procedures at the server

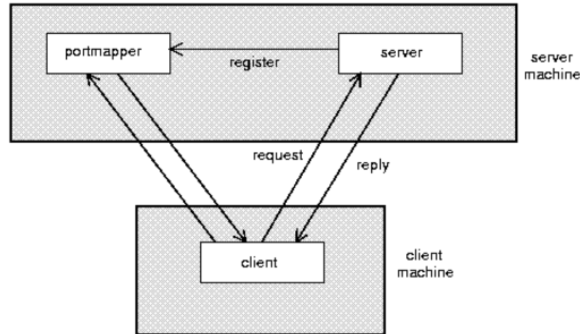


Figure 4.10: Port Mapper

4.7 Python Remote Objects (PyRO)

Basically an RMI for Python objects. It is a library that enables you to build applications in which objects can talk to each other over the network, with minimal programming effort. You can just use normal Python method calls to call objects on other machines.

Steps involved in using PyRO for Python RMI -

1. In server code,
   (a) PyRO daemon instantiated
   (b) Remote class registered aas PyRO object
   (c) Get URI so we can use it in the client later
   (d) Start the event loop of the server to wait for calls
2. Start server
3. In client code,
   (a) Get a Pyro proxy to the remote object using its URI
   (b) Call method normally
4. Start client (from remote machine)

Q: How is the client supposed to get the uri, if we're not copy-pasting it?
A: Pyro provides a name server that works like an automatic phone book. You can name your objects using logical names and use the name server to search for the corresponding uri.

4.8 gRPC

gRPC is Google's RPC platform, developed for their internal use but which is now open-source and available to all developers. It is a modern, high-performance framework for developing RPCs, which was designed for cloud based applications. gRPC is designed for high inter-operability - it works across OS, hardware,

and programming languages. Client and server do not have to be written in same langugae, gRPC supports multiple languages including (python, java, C++,C#, Go, Swift, Node.js, etc). It uses http/2 as transport protocol, and ProtoBuf for serializing structured messages. Http/2 is more efficient than TCP/IP, and ProtoBuf allows for interoperability.

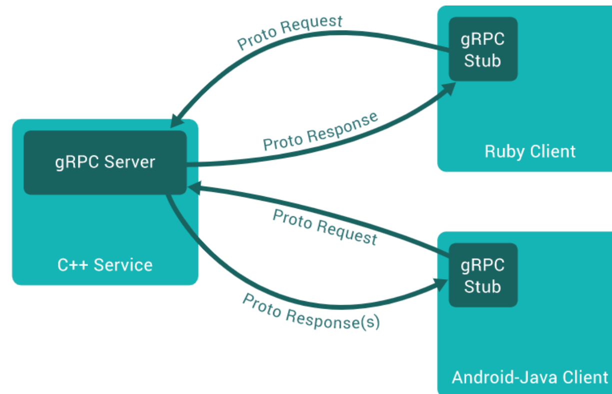


Figure 4.11: gRPC

4.8.1 Protocol Buffer (ProtoBuf)

ProtoBuf is a way to define a message and send it over a network which can be reconstructed at the other end without making any assumptions about the language, OS, or hardware used (platform independent). ProtoBuf has marshalling/serialization built-in.

A ProtoBuf message structure is defined in a `.proto` file (see Fig 4.12). It uses protocol compiler protoc to generate classes. Classes provide methods to access fields and serialize/parse from raw bytes e.g., set_page_number(). It is similar to JSON, but in binary format and more compact.

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

Figure 4.12: ProtoBuf Message Structure

Q: How does ProtoBuf handle objects?
A: ProtoBuf cannot send code, objects contain code. An object written in Java would not make sense in another language like Python. However you can send arbitrary data structure like arrays, vectors, hashmaps etc.

4.8.2 gRPC Example

1. Define gRPCs in proto file with RPC methods

- params and returns are protoBud messages
- use protoc to compile and get client stub code in preferred language

2. gRPC server code on server side

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Figure 4.13: gRPC Example

4.8.3 gRPC Features

- Four types of RPCs supported -
  1. Unary: Receive one request and send back single response
  2. Server Streaming: Server can return stream of responses.
  3. Client Streaming: Client can return stream of requests.
  4. Bi-directional: Client and server can both send stream of messages to each other.
- Supports synchronous and asynchronous calls
- Deadlines/timeouts: client specifies timeout, server can query to figure out how much time is left to produce reply
- Cancel RPC: server or client can cancel RPC to terminate it

Lecture 5: February 9

Professor: Prashant Shenoy *Scribe: Madhuvanthi Ramesh Kumar (2022), Josh Sennett (2019)*

5.1 Overview

This lecture covers the following topics:

Part 1: Threads

Part 2: Concurrency Models

Part 3: Thread Scheduling

5.2 Lecture Notes

5.2.1 Part 1: Threads and Concurrency

5.2.1.1 Review of Processes and Threads

A **process** is a program that is executing on a system. Each process has its own address space with its corresponding code, global and local variables, stack, and resources.

A **thread** is a traditional process that has multiple concurrent flows of control. Each flow of control executes a sequence of instructions. Multiple threads can be part of the same process and can be executed concurrently. All threads share the same address space, but have their own stack and control flow as they execute different parts of the process. Synchronization might be required if threads are accessing shared data structures.

Question: When there are multiple threads, in what order will they execute?
Ans: Order of execution depends on order of scheduling, which is done by OS, using various scheduling algorithms.

Concurrency vs Parallelism
Concurrency enables handling of multiple requests. Multi-threading can be used to achieve concurrency if we are using a machine with a single core. The different threads are time multiplexed onto the processor, and different parts of the program can be executed by switching between threads.
Parallelism enable simultaneous processing of requests. Multi-threading can be used to achieve **parallelism** on a multi-core machine, as two threads can simultaneously execute in parallel on different cores.

Question: Does the process decide how much memory is allocated to each thread?
Ans: The OS decides how much memory to give each process. Thread runtime decides how much stack space to allocate to each thread. The heap is shared across threads.

5.2.1.2 Why use threads?

Multi-threading allows for concurrency within a single process. On a a machine with a single processor, threads achieve concurrency through time-sharing of the CPU. While one thread is doing I/O, another thread can execute on the CPU, thereby improving performance. On a multiprocessor machine, we can achieve true parallelism by running threads simultaneously on different cores.

Switching between threads of the same process or different processes is more efficient than switching between processes. Thread operations such as thread creation, deletion, switching are much more efficient and lightweight than the corresponding process operations.

Threads have access to the entire address space of the process, which gives programmers greater flexibility, allowing for shared data rather than message passing.

If a single threaded process makes a blocking call, the entire process is blocked till the blocking operation is completed. If you have multiple threads within that process, each thread is a synchronous sequential stream of execution. If one thread makes a blocking call, other threads of the process can continue executing.

In between single and multi-threaded programming, there is **finite-state machine** (event-based) programming. Event-based programming attempts to achieve concurrency with a single threaded process, using non-blocking calls and asynchronous communication (which is more complex to program).

5.2.1.3 Use cases - Client and server

An example for a multi-threaded client would be a Web Browser. Each task that a browser performs could be assigned to a different thread. If the browser was single threaded, upon clicking on a web link, images would have to be sequentially downloaded from the server and then sent to be parsed and rendered. In a multi-threaded browser, images can be downloaded in parallel while the page is parsed and parts of the page can be rendered as they are ready. The browser does not have to wait for everything to be downloaded and parsed before rendering. It can connect to different servers and exchange data using different threads. As a result, the user will see parts of the page more quickly.

Multi-threading is also used within servers. If a server only runs a single thread, it might have a queue of requests from clients which are blocked until the first request is completed. Using a pool of threads allows the server to respond to multiple requests simultaneously, thereby reducing latency. The idea is for the server to have a dispatcher and a few worker threads. When a client request comes in, the dispatcher assigns this request to one of the idle worker threads. This model is efficient because, while some of the worker threads are I/O bound, others can continue doing computation.

Question: If there's a long sequential piece of code, is there any benefit to dividing it into smaller pieces?
Ans: If the smaller pieces are independent tasks, you can put them in threads and they will execute concurrently (single processor) or even in parallel (multiprocessor). If the tasks are dependent, you can't parallelize it as there is a dependency.

Question: What is concurrency?
Ans: If there are multiple threads within a process, and a single core, one thread will execute for a small time slice, and then switch to another thread for the next time slice, and so on. This is known as concurrent execution.

Question: If there is a program that has no blocking calls, running on a single core machine, will having multiple threads reduce the time needed for execution?
Ans: Multi-threaded program does not change the time required to execute instructions. It only interleaves

the instructions.

5.2.2 Concurrency Models

All server applications involve a loop, called the event loop, to process incoming requests. Inside this loop, the server waits for an incoming request from the client over a network and then processes the request.

5.2.2.1 Sequential Server

It is a single threaded server that processes incoming requests sequentially.
Advantage: It is very simple to implement.
Disadvantage: If a burst of requests arrive, they queue up at the server. As the server processes requests sequentially, requests will queue up while one request is being processed. This increases the waiting time and response time.

5.2.2.2 Multi-threaded Server

- **Thread per request**
  In this model, every request is assigned to a new thread. The thread processes the request, and is then terminated.
  Advantage: Newly arriving requests don't need to wait for older requests to be processed.
  Disadvantage: Frequent creation and deletion of threads causes overhead.

  Question: Is the time to finish a request longer because you need to share time with other requests?
  Ans: The response time depends on what kind of processing the request involves. If the processing involves I/O, you can process another request if you have multiple threads. If the request involves CPU processing, by interleaving multiple requests, the request processing time will increase due to switching. However, it is still better overall as wait times could be very high in a sequential model.

  Question: How many threads can an application have? Is it limited by number of cores? **Ans**: There is no restriction on number of threads. However, there is an overhead as each thread involves maintaining some state, which requires memory.

- **Thread pool**
  In this model, a fixed number (N) of threads are created when the server is started. This is known as a thread pool. One thread acts as the dispatcher and the other threads are the workers. The dispatcher pulls a request from the queue and assigns it to an idle thread to process.
  Advantage: We do not need to create and delete threads for each request.
  Disadvantage: It is difficult to choose the right N. If more than N requests arrive concurrently, there will be blocking till a thread becomes idle. If we have a very large thread pool, resources will be wasted as many threads will be idle.

  Question: Is there any interleaving between idle threads?
  Ans: If a thread is waiting, it will not be scheduled for execution. Only active threads get time slices and are involved in the switching.

- **Dynamic Thread pool**
  As we saw in the previous model, it is often difficult to choose the optimal pool size as it depends on the rate of incoming requests. In this model, we have a dynamic pool where we start with N threads

and monitor number of idle threads. If the number of idle threads fall below a certain threshold, we increase N and start new threads. If the number of idle threads is greater than a threshold, we reduce the size of the thread pool and terminate some idle threads. We can configure the size of the pool, and even set a max and min limit. This model is pro-active as it creates/deletes threads before it reaches the min/max.
Advantage: We need not worry about choosing an optimal pool size.
Disadvantage: Server is more complicated, requires monitoring and adjustment of pool size dynamically.

- **Asynchronous Event Loop**
  It is a single threaded server, but uses non-blocking operations. It provides concurrency similar to synchronous multi-threading, but with a single thread. If a blocking operation needs to be executed, we switch to some other operation and let the blocking operation continue working in the background. Switching between parts is driven by code, rather than the OS.

  Question: How does the program know when a blocking call has finished executing?
  Ans: Standard I/O operations such as read are blocking. Asynchronous read sends the request to the OS and the control returns back to the process. When the read finishes, the OS sends a signal to the process as an event, and the process goes back to what issued the asynchronous call and continues execution from there.

  Note: If you make a function async, you need to make all blocking calls inside the function also async.

- **Process Pool Servers**
  These are multi process servers that use a separate process to handle each request. This can be done by creating a (dynamic) process pool. As there are multiple processes involved, this model requires inter-process communication.
  Advantage: If a process crashes, other processes continue to run. Crashes only impact that request, not the entire application. In a multi-threaded server, all threads are part of the same process. If one thread crashes, the entire server crashes.
  This model enables address space isolation. Sensitive operations can be carried out as one process doesn't have access to another process' memory.
  Disadvantage: More expensive, process switching is more heavyweight than thread switching.

5.2.2.3 Summary

Based on architecture, servers can be broadly classified into four categories:

1. Pure sequential
2. Event based
3. Thread based
4. Process based

Single threaded event based servers have high efficiency, but are the hardest to program. If there are multiple cores, multi-threading is more efficient as you can achieve true parallelism. For best performance, we need a multi-threaded event based architecture.

5.2.3 Part 3: Thread Scheduling

5.2.3.1 User-Level Threads

User-level threads are created and managed by user libraries. The kernel is unaware of there being multiple threads. The kernel sees the address space of the threads as a traditional single-threaded process.

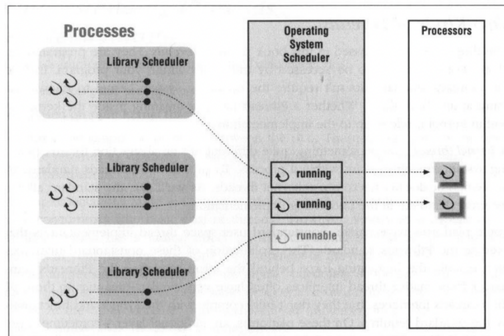


Figure 6-1: User-space thread implementations

Advantage: Creation of threads is very lightweight because it doesn't require system calls.
Disadvantage: It cannot take advantage of parallelism. As the OS sees only a single thread, even if there are multiple threads, they cannot run on multiple cores.

Question: How do you write a user level thread library?
Ans: The library has to use event based programming. It needs to see threads as being blocks of code, and switch between them. Need to provide the same APIs.

5.2.3.2 Kernel-Level Threads

Kernel-level threads are created and managed by the OS. Kernel can see the presence of threads and can schedule them.

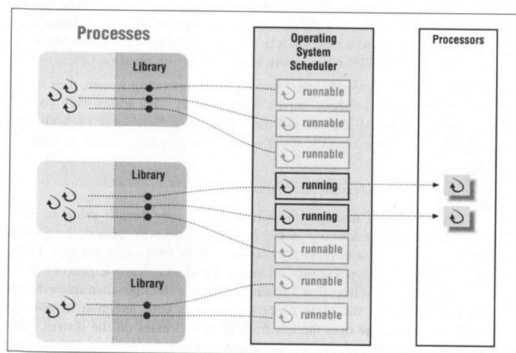


Figure 6-2: Kernel thread–based implementations

Advantage: They allow for real parallelism between threads as the OS is aware of multiple threads and can run different threads on different cores.
Disadvantage: They are more expensive as thread management is handled by the OS.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 6: February 14

Lecturer: Prashant Shenoy *Scribe: **Mrinal Tak***

6.1 Overview

In the previous lecture, we learned about the different kinds of threads, specifically user-level threads and kernel-level threads. In this lecture we will cover:

Thread Scheduling

Scheduler Activations

Lightweight process

Multiprocessor scheduling

Distributed scheduling

Case Studies : V-System, Sprite, Volunteer Computing, Condor

6.2 Thread Scheduling Example

Here, we assume that CPU scheduler uses round-robin time slices. We are given that the program consists of two functions: foo() and bar().

Case 1. No I/O involved: For a single threaded process, sequential execution will first complete the execution of foo() and then after the completion of foo(), the execution of bar() starts. In case of multithreaded process, two separate threads will execute the functions foo() and bar() concurrently and the execution will be interleaved. In terms of completion time, there will be no improvement.

Case 2. I/O involved: Here we will see the advantage of running a multi-threaded process. In case of sequential run, the CPU will be idle and there will be no computation. But in case of multi-threaded process, execution time will shrink, as I/O does not block CPU. If we run the multi-threaded process on multiple cores, the I/O does not help us run extra computation, but the execution time shrinks due to true parallelism.

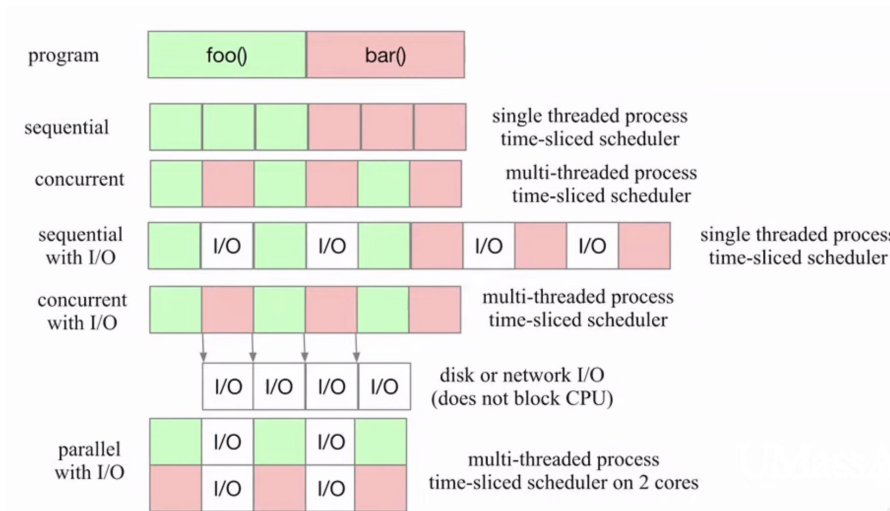


Figure 6.1: Example of thread scheduling.

6.3 Scheduler Activations

Scheduler activations are a threading mechanism which give the performance of user-level threads with the behavior of kernel threads. There is active cooperation and exchange of information between the user-level thread scheduler and the OS to get better scheduling. The same idea is also present in lightweight processes.

6.4 Lightweight Processes

Lightweight processes are a schedulable entity which exist on a layer between kernel-threads and user-threads. They are managed by the kernel. One actual process may use multiple LWPs. Each LWP is bound to a kernel thread. User-level threads can be flexibly mapped to LWP. The mapping is decided by the developer and can be $n : m$, where n threads in user-level are mapped to m threads in kernel-level.

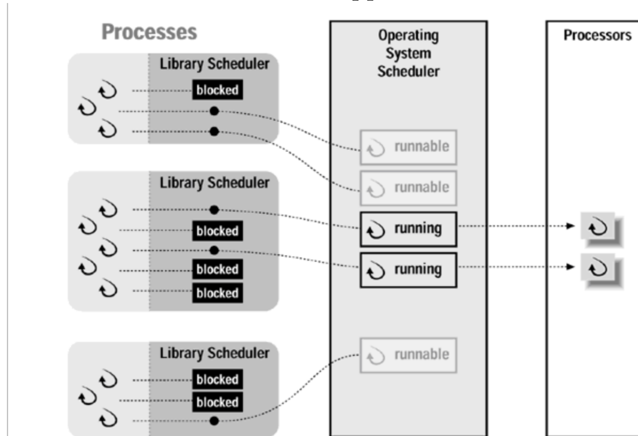


Figure 6.2: Lightweight processes.

6.5 Multiprocessor scheduling

Here, we will talk about single machines with shared memory multiprocessor or multi-core CPU. Looking at the diagram below, the circles at the top are processors. Caches are presented at each of the processors. They keep instructions or data, and are used to speed up the execution of the program. There might be more than two level of caches (L1, L2, L3). Some caches are shared; others are not. Memory, or RAM, are shared across all of the processors using system bus; a program running on one processor can access any address in memory. Multi-processor scheduling involves scheduling tasks in such environment.

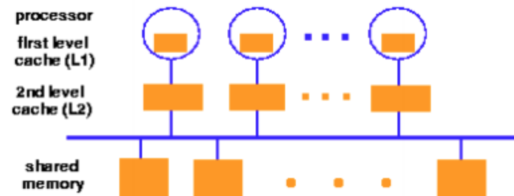


Figure 6.3: Multiprocessor scheduling.

6.5.1 Central queue implementation

In a central queue, all of the processors share a single global queue where all the threads and processes are present. Whenever the time slice on any processor is going to end, that processor will look at the ready queue, pull a thread or process from that queue, and schedule it.

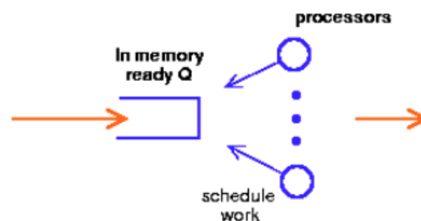


Figure 6.4: A central queue.

6.5.2 Distributed queue implementation

In a distributed queue, there are more than one queue in the system. The processes or threads are going to be part of one of those queue. When a processor becomes idle it is going to look at its local queue, and only take the next job in that queue to schedule for execution.

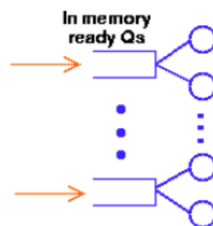


Figure 6.5: A distributed queue.

6.5.3 Pros and cons of centralized queue and distributed queue

The centralized queue is just shared data structure. As the number of processors grows, we face the problem of **synchronization bottleneck**. We have to use some synchronization primitives - the standard one being a lock. When the time slice on one processor ends, that processor has to go to the queue, lock the queue, and pick a job. If another processor becomes idle during this, this processor has to wait until the previous processor has picked the job and released the lock. No matter how efficient synchronization primitives are, there is still going to be synchronization overhead. That overhead will grow as the number of processors increases because there is going to be **lock contention**. There will be a lot of idle timeslices which will be waster.

Distributed queue experiences less contention. When there is one queue per processor, there is hardly contention at all. The processor just goes to its local queue and pull a thread. Multiprocessor scheduling wouldn't be impacted by the # of processors in a system.

But distributed queue needs to deal with **load imbalance**. Suppose there are n tasks. The n jobs will be split across the distributed queues. How splitting occur matters - if the queues are not equally balanced then some tasks are going to get more cpus time than others. Tasks in shorter queues get more round-robin timeshare. To deal with this, every once in a while look at all the queues and their lengths, and equalize them again so that every process gets approximately fair share of CPU time.

Cache affinity is important. Respecting cache affinity can hugely improve time efficiently. A process or thread has affinity to a processor because the cache there holds its data. Let's say processor 1 picks up a job and schedules it for a time slice. By the time that time slice ends, the cache for that processor would be warm, with data and instruction for that job stored in the cache. The process goes to the end of the queue, eventually appears at the front, and get scheduled at another processor. There is no mapping of task to processor in the centralized queue. That processor will start from a cold cache, with no data and instruction for this job. The initial instructions will all be cache misses. Program execution is slowed down to warm up the cache, and then time slice ends.

This argues using distributed queue based scheduling. Once process joins a queue it will stay in the same queue. Next time it gets scheduled there will be data and instructions from the previous time slice i.e. start with warm cache. But even though you might try to schedule process or thread on the same processor where it was executed last, every time you run again you still might not have all your data and instructions in the cache due to other processes in your system. You want to have larger time slices or quantum durations to account for the time when there is some early fraction of the quantum maybe all caches misses. You want to still get to run on the caches once it's warmed up for some period of time. As a result, time slices in multiprocessor systems tend to be longer than the ones in uniprocessor systems.

While designing CPU scheduler, Cache Affinity plays the most important part. It should be considered over synchronization bottleneck and lock contention. So, there are couple of things to keep in mind while scheduling on multiprocessors: 1. Exploit cache affinity: Try to schedule on the same processor that a process/thread executed last. 2. Pick larger quantum sizes for the time slicing to decrease context switch overhead.

6.5.4 Scheduling parallel applications on SMP using gang scheduling

One last point on multiprocessor scheduling. Until now, we assumed that the processes/threads are independent. We just picked the one at the front of the queue and we did not care about what was running in other

processors. If you think of a parallel process with many many threads or a job with many processes that are coordinating some larger activities, you might not want to schedule them independently. For example, let's say there are two cpu's and a process with two threads. Let's assume when these threads are executed they also need to communicate with each other for application needs. When T1 runs on Processor A, T2 may not be running on Processor B i.e. some other job may be running on Processor B. If T1 sends a message to T2, then it will wait for a reply until processor B runs T2 (so T2 can actually receive the message and process it). There is increase in waiting time.

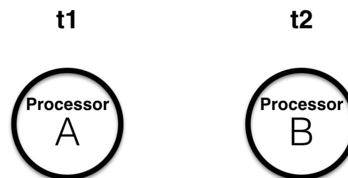


Figure 6.5: Two cpu, and a process with two threads.

Gang scheduling schedule parallel jobs to run on different processors together, as a group in the same time slice. This allows true parallelism, i.e. all the threads of a process can run simultaneously. Gang scheduling is used in special-purpose multi-processors that are designed specifically to execute specialized, massively parallel applications such as scientific jobs.

If one component blocks, the entire applications will be preempted. The remaining $n-1$ timeslices will end and the all the gang components will resume when the other thread gets unblocked. Smart scheduling can take care of it too, by adjusting the priority.

Q: Is there a general purpose scheduling that you would normally run on a parallel machine and switch to gang scheduling only when there is a parallel application?

A: That is technically possible, but the hardware machines that gang schedulers run on are so specialized you would not run general-purpose applications.

6.6 Distributed scheduling

6.6.1 Motivation

Consider the scenario in which we have N independent machines connected to each other over a network. When a new application or process arrives at one of the machines, normally the operating system of that machine would execute job locally. In distributed scheduling, you have additional degree of flexibility. Even though the user submit job at machine i, the system may actually decides to execute the job at machine g and bring back the results. Distributed scheduling - taking jobs that are arriving at one machine and running them somewhere else in the system - does this makes sense (any advantage to this)?

Scenario 1: Lightly loaded system

You should run job locally at the machine where it arrives. No benefit of moving a job because you have enough resources to do it.

Scenario 2: Heavily loaded system

You want to move a job somewhere else but there are no resources available in the system for you to run that job.

Scenario 3: Moderately loaded system

If a job arrives at machine i, and machine i happens to be slightly less loaded than other machines, then run it locally. If machine i happens to be more heavily loaded than other machines, then find another machine to run that job. This scenario would actually benefit from distributed scheduling.

A more technical phrasing of the question would be: *what's the probability that at least one system is idle and one job is waiting?* System idle means that exists some machine that has more resources to actually run more jobs than it is currently running. Job waiting means there is a job that arrived at heavily loaded machine and waiting to run. We benefit from distributed scheduling when both of these cases are true.

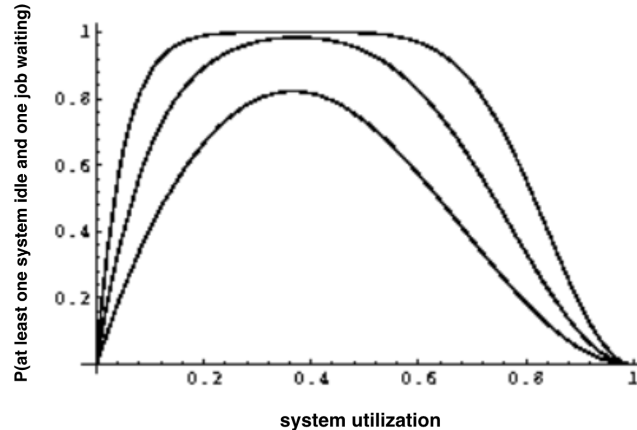


Figure 6.6: The relationship betweem system utilization and resource undertilization.

- Lightly loaded system:
  P(at least one system idle) high
  P(one job waiting) low
  P(at least one system idle and one job waiting) low
- Heavily loaded system:
  P(one job waiting) high
  P(at least one machine idle) almost zero
  P(at least one system idle and one job waiting) low
- Medium loaded system is the area of high probability under the curve

Note: P(at least one system idle and one job waiting) = P(one job waiting) * P(at least one system idle)

The three lines are constructed by different system parameter settings but the overall curve is the point.

6.6.2 Design issues

Performance metric: mean response time.

Load: CPU queue length and CPU utilization. Easy to measure and reflect performance improvement.

Types of policies:

- In static policy, decisions are hard-wired into scheduling algorithm using prior knowledge of system.
- In dynamic policy, the current state of the load information is used to dynamically make decisions.
- In adaptive policy, parameters of the scheduling algorithm change according to load. Can pick one of static or dynamic policy.

Preemptive vs. Nonpreemptive:

- Preemptive – once job has started executing, scheduler can still preempt it and move somewhere else. Transfer of a partially executed task is expensive due to collection of task's state.
- Nonpreemptive – only transfer tasks that have not begun execution.
- Preemptive schedulers are more flexible but complicated.

Centralized vs. Decentralized:

- Centralized – queue makes decision to send a job globally.
- Decentralized – queue makes decision locally.

Stability: In queuing theory, the arrival rate should be less than system capacity or else the system will become unstable. When arrival rate is reaching system capacity, machines don't want incoming jobs and sends jobs off to other machines which also doesn't want them. Lots of processes are being shuffled around and nothing gets done. Queue starts building up, job floats around, system gets heavily loaded.

6.6.3 Components of scheduler

Distributed scheduling policy has 4 components to it.

i. **Transfer policy** determines **when** to transfer a process to some remote machine. Threshold-based transfer policies are commonly used to classify nodes as senders, receivers, or OK.

ii. **Selection policy** determines **which** task should be transferred from a node. The simplest is to select a newly arrived task that have caused the node to become a sender. Moving processes that have already started executing is more complicated and expensive. The task selected for transfer should be such so that the overhead from task transfer is compensated by a reduction in task's response time.

iii. **Location policy** determines **where** to transfer the selected task. This is done by polling (serially or in parallel). A node can be selected for polling randomly, based on information from previous polls, or based on nearest-neighbor manner.

iv. **Information policy** determines when and from should above information of other nodes be collected - demand-driven, periodic, or state-change-driven - so that the scheduler can make all of its decisions in the right way.

6.6.4 Sender-initiated distributed scheduler policy

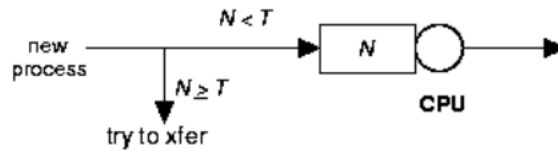


Figure 6.7: A sender-initiated distributed scheduler policy.

Overloaded node attempts to send tasks to lightly loaded node.

- **Transfer policy**: CPU queue threshold, T, for all nodes. Initiated when a queue length exceeds T.
- **Selection policy**: newly arrived tasks.
- **Location policy**:
  - Random: select a random node to transfer the task. The selected node may be overloaded and need to transfer the newly arriving task out again. Is effective under light load conditions.
  - Threshold: poll nodes sequentially until a receiver is found or poll limit has been reached. Transfer task to the first node below threshold. If no receiver is found, then the sender keeps to the task.
  - Shortest: poll N_P nodes in parallel and choose least loaded node below T. Marginal improvement.

  There are more sophisticated approaches of finding the node, as the above trivial approaches are not scalable for a high number of nodes. One such approach is, every machine keeps a table which stores the load of other machines. Whenever load on a machine changes, it updates the tables stored in other nodes. In this case the location policy is just a table lookup. The other way to achieve it is to elect a coordinator which maintains the loads of all other nodes, so now there is a central table which maintains the load of other nodes.
- **Information policy**: demand-driven, initiated by the sender.
- **Stability**: Unstable at high-loads

6.6.5 Receiver-initiated distributed scheduler policy

Underloaded node attempts to receive tasks from heavily loaded node.

- **Transfer policy**: when departing process causes load to be less than threshold T, node goes find process from elsewhere to take on.
- **Selection policy**: newly arrived or partially executed process.
- **Location policy**:
  - Random: randomly poll nodes until a sender is found, and transfer a task from it. If no sender is found, wait for a period or until a task completes, and repeat.
  - Threshold: poll nodes sequentially until a sender is found or poll limit is reached. Transfer the first node above threshold. If none, then keep job.
  - Shortest: poll n nodes in parallel and choose heaviest loaded node above T.
- **Information policy**: demand-driven, initiated by the receiver.
- **Stability**: At high loads, a receiver will find a sender with a small number of polls with high-probability. At low-loads, most polls will fail, but this is not a problem, since CPU cycles are available.

6.6.6 Symmetrically-initiated distributed scheduler policy

Sender-initiated and receiver-initiated components are combined to get a hybrid algorithm with the advantages of both. Nodes act as both senders and receiver. Average load is used as threshold. If load is below the average load, node acts as receiver. If load is above average threshold, node acts as sender.

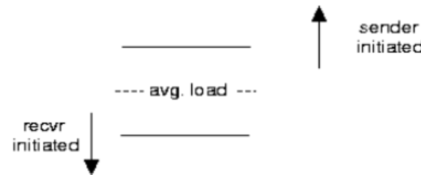


Figure 6.8: A symmetric policy.

Improved versions exploit polling information to maintain sender, receiver, and OK nodes. Sender polls node on receiver list while receiver polls node on sender list. The nodes with load above Upper Threshold(UT) becomes a sender nodes, whereas nodes with load below Lower Threshold (LT) become the receiver nodes.

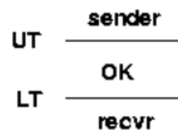


Figure 6.9: An improved symmetric policy.

Q: Isn't there communication overhead?

A: Some; can broadcast your node and keep a table every minute or every time load changes.

Q: How to pick threshold?

A: That's configurable parameter; policy doesn't care.

6.7 Case Study

6.7.1 V-System (Stanford)

Following a state-change driven information policy, the V-System broadcasts information when there is significant change in CPU/memory utilization. Nodes would listen to the broadcast and keep a load table.

The V-System also implements a sender-initiated algorithm which maintains a list of M least loaded nodes. While off-loading, it probes a possible receiver randomly from M and transfers job only if it is still a receiver. When locating a node to receive job transferal, the system needs to double check whether the node on the list of M least loaded nodes is still a receiver because the load table is typically not updated super frequently unless policy did on-demand polling. One job is off-loaded at a time.

6.7.2 Sprite (Berkeley)

Sprite assumes a workstation environment in which ownership is king. When users are on their desktop they own it, and no other tasks will run on it. Other tasks can only run on desktops when there is no user using it. Like V-System, Sprite also uses state-change driven information policy and implements a sender-initiated scheduling algorithm.

There is a **centralized coordinator** which keeps the status of load on all the machines. They made an assumption that if there is no mouse or keyboard activity for 30 seconds and the number of active processes is less than a threshold meaning there is at least one processor idle the node becomes a receiver. Foreign processes get terminated when the user returns.

Sprite implemented **process migration** so that the process can suspend running job on a node and continue running from its last state at some other node, instead of killing and restarting the job. Process migration first stops a running job, writes out all registers and memory contents to disk, and transfers the entire memory contents as well as kernel state to receiver machine. Process migration is fairly complicated and comes with many problems e.g. I/O, network communication. Sprite comes at restriction and can only migrate certain types of process.

How does process migration works? It says process is a program executing in memory. To migrate the process, take the memory contents and move it to another machine. Since there is no shared memory, we use the distributed file system as intermediary.

Steps:

1. Suspend the execution of the process, so that its memory contents no longer change.

2. Copy the contents from memory to disk.

3. On the other machine, start paging in. Since the file system is shared and not the memory, we can load the process into receiver's memory.

Q: Are the jobs exclusively CPU, or they can do I/O too?

A: They can do I/O. There is filesystem shared across all machines in some centralized server.

Q: What if everybody goes to lunch (leave the machine idle) and comes back to work (use machine) at the same time?

A: There can be a queue of jobs awaiting execution, so they can be run on idle nodes when users are away.

6.7.3 Condor (U. of Wisonsin)

Condor makes use of idle cycles on workstations in a LAN. It can be used to **run large batch jobs** and long simulations. It has a central job management system, called the **condor master**, which idle machines contact to get assigned waiting jobs. It supports process migration and flexible job scheduling policies. The SLURM scheduler on UMass Swarm cluster is an example of this paradigm.

Condor is not an OS, it is a software framework which runs over OS.

6.7.4 Volunteer Computing

Volunteer computing is based on the idea that PCs on the internet can volunteer to donate CPU cycles/storage when not in use and pool resources together to form an internet scale operating system. A coordinator partitions a large job into small tasks and sends them to the volunteer nodes.

Reliabilty is an issue here. So instead of relying on a single machine, the same subtask in run on k different machines and it is made sure that the answer matches. Seti@home, BOINC and P2P backups are examples for this paradigm.

Lecture 7: February 16

Lecturer: Prashant Shenoy *Scribe: Shruti Jasoria (2022), Deokgi Hong (2019), Ao Liu (2018)*

7.1 Overview

In this lecture, we continued Cluster Scheduling and started a new topic, Virtualization.

7.2 Cluster Scheduling

Cluster Scheduling is scheduling tasks on a pool of servers. This differs from distributed scheduling on workstations. We assume:

- Machines are cheap enough that looking for free CPU cycles on an ideal workstation is not required. There is a dedicated pool of servers to run the tasks.
- Servers are more computationally more powerful than workstations.
- The users explicitly submit the jobs on the cluster.

Cluster scheduling can be analyzed with respect to two applications:

- **Interactive Applications**
  These are applications where user has sent a request and is actively looking for a response. In this case response time is an important factor in designing the scheduler, e.g., for web services.
- **Batch Applications**
  These are long running computations which require powerful machine. In this case the throughput is optimized, e.g., for running simulations.

7.2.1 Typical Cluster Scheduler

Typically, there are two kinds of nodes in a cluster - dispatcher and worker. A request (or task) follows the following process:

1. The user submits a request to the dispatcher node.
2. The dispatcher node places the task on a queue.
3. When a worker node is available, the dispatcher removes the task from the queue and assigns it to a worker node.
4. The worker node runs the task.

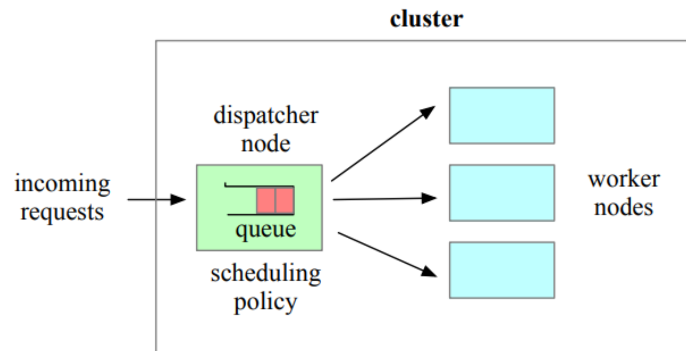


Figure 7.1: A typical cluster scheduler

This design is similar to a thread pool that has listener and worker threads. The scheduling policy depends on the kind of application that has to be run on the cluster.

Question: Would the dispatcher act as single point of failure?
Answer: That would always be a case. To make the system failure tolerant it has to be ensured that in case the dispatcher fails another machine can take over. [1]

7.2.2 Scheduling in Clustered Web Servers

Interactive applications like websites that receive large number of requests are hosted on a cluster. Consider a cluster of N nodes. Here one node acts as the dispatcher, called the *load balancer*. The $(N-1)$ worker nodes are running a replica of the web server and act as the server pool. Any of the worker nodes can service the user requests. The dispatcher receives the request from the user and directs it to one of the worker nodes. The scheduling of the user requests can be done on the basis of:

- The dispatcher can pick up the **least loaded** server.
- The dispatcher can schedule the job in a **round robin** manner. In this case it would not need to get periodic load information updates from the worker nodes.
- If the worker nodes are heterogeneous, **weighted round robin** can be used where you weight a node by the amount of resources and allocate more requests to machines with more computation power.

In case of stateful applications, for eg. an online shopping store the state of the shopping cart is maintained by the machine which served the user request. In this case the request level scheduling won't work as the worker node maintains the session for the user. In these cases **session-level load balancing** is used. When a new web browser starts a session round robin is used to allocate the request to worker node. The server is then mapped to the user and all the subsequent requests by the user are sent to the same machine. One way around this is sending the state back to the client as a cookie which the client has to send it back to the server with every request. This way the state is maintained on the client and not the server. This can allow request level load balancing but would be more expensive as the state updates are sent back to the user.

Question: If you have state, can you keep it in shared memory or database?

[1] A good case study on how this can be achieved is Apache Kafka.

Answer: Shared memory is not a typical OS abstraction. There is no such thing as distributed shared memory that most operating systems support. There can certainly be distributed storage or shared database that can be used to maintain the state. To do so, you'd have to pay the price of I/O to retrieve the state for every request. [2]

7.2.3 Scheduling Batch Applications

This is used for larger jobs like ML training, data processing, simulation. These are long running jobs that can take seconds to as long as hours to finish running. All these jobs come to the dispatcher queue and have to be assigned to a worker node. For this we need a batch scheduler that decides which node to run the job.

One popular batch scheduler available on linux muchines is **SLURM** (Simple Linux Utility for Resource Management). Fundamentally it has a queue and it has a scheduling policy. But it has many other feature:

- It divides the cluster into subgroups. Each subgroup has its own queue.
- This allows user groups with separate scheduling policies for jobs.
- The various queues can put constraints based on runtime of the job. For eg. a queue for short jobs would terminate the job if exceeds the time limit.

7.2.4 Mesos Scheduler

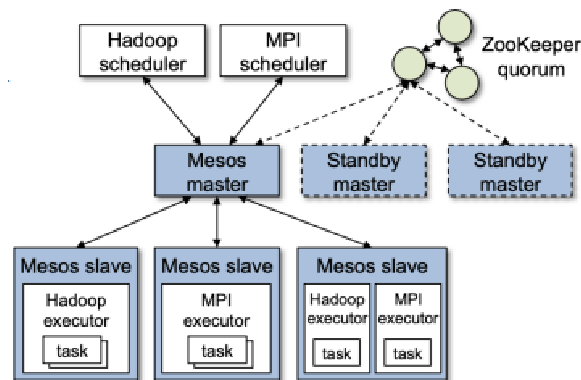


Figure 7.2: Mesos Architecture

The Mesos Scheduler was designed in Berkley but now has been adopted in open source projects. It is a cluster manager and scheduler for multiple frameworks. Mesos dynamically partitions the cluster on the fly based on resource needs of different frameworks. It has a heirarchial two level approach. Mesos allocates resources to a framework and the framework allocates resources to the task. At any point, Mesos keeps taking back resources from frameworks that do not require them anymore and assigns them to frameworks that need them.

Mesos introduces the concept of **resource offer**. Whenever there are idle servers, Mesos creates an offer and sends it to the framework. The framework either accepts or rejects the offer made by Mesos. It accepts the offer if the resources being allocated are sufficient to run the task. Otherwise, it rejects the offer. There

[2] For example, Redis can be used as a distributed in-memory cache because of

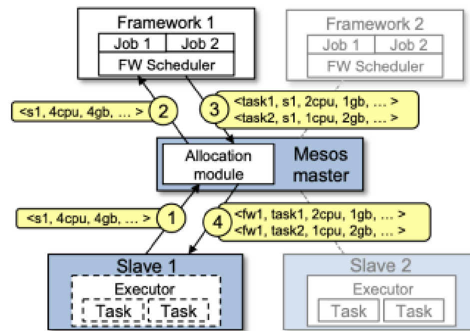


Figure 7.3: Mesos Scheduling

are policies to decide which framework has to be offered the resources. Once the task run by framework is done and the resources are idle, it is available to be re-offered by Mesos.

Question: If there is a busy framework and a not-busy framework, do you want to offer the a not-busy framework more resources?
Answer: The ramework itself is a cluster scheduler. If the framework is busy, it'll take the resources. It is advantageous for the framework to get more resources.

Follow up question: Would a framework accept resources even if it has satisfactory amount of resources?
Answer: A framework accepts resources if it has pending tasks in its queue. The frameworks are designed to be well-behaved, i.e., they will not accept resources if they don't need them.

Question: Once the resources are free, how are the sent back to Mesos master?
Answer: The worker nodes have a mesos monitor running on them. Once the node is idle, that entity returns the node to Mesos master.

Question: Does Mesos have any intervention if the client contacts the server?
Answer: From the client's perspective, Mesos is completely transparent. The client submits the jobs to the queue of relevant framework.

Question: What is the advantage of Mesos?
Answer: The pool of servers allocated to a framework can be dynamically changed at a server granularity. This allows adjustment of resources based on the load on each framework. This makes better utilization of servers in the pool.

Question: Does the Mesos coordinator know resource requirement of each framework?
Answer: No, it doesn't. That is why it makes offers. Mesos makes its decision based on acceptance or rejection of the offer. If the offer is rejected it could be one of the two situations. Either the framework doesn't require more resources or the resources offered are not enough. Mesos would probe and try to figure out the situation. It's a "push" approach.

Question: When a task finishes the idle server goes back to the master. Is it a good idea to give up the resource as another task could be received by the framework and that could have run on the idle node?
Answer: Server allocation is done on a per-task basis. If a task is waiting, the framework would eventually get resources assigned.

7.2.5 Borg Scheduler

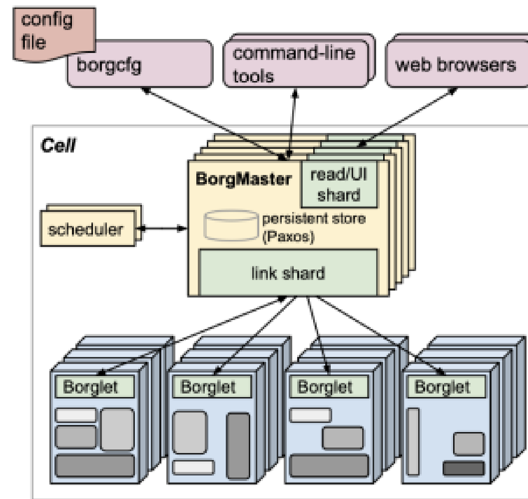


Figure 7.4: Borg Scheduler

This is Google's cluster scheduler that was designed with the following design goals:

- The culture should scale to hundreds of thousands of machines.
- All the complexity of resource-management should be hidden from the user.
- The failures should be handled by the scheduler.
- The scheduler should operate with high reliability.

These ideas were later used to design Kubernetes. Borg is designed to run both interactive and batch jobs. Interactive jobs like web search, mail are user facing production jobs are given higher priority. Batch jobs like data analysis are non-production jobs are given lower priority. Interactive jobs will not take up the entire cluster. The remaining cluster is used for batch jobs.

In Borg, a **cell** is a group of machines. The idea behind Borg scheduler is to match jobs to cells. The jobs are going to specify the resource needs and Borg will find the resources to run it on. If a job's needs change over time, it can ask for more resources. Unlike Mesos, the jobs do not wait for an offer but "pull" the resources whenever they need it.

Borg has built-in fault tolerance techniques. For e.g. if a cell goes down, Borg would move the job to some other cell. To allocate set of machines to a job, Borg would scan machines in a cell and start reserving resources on various machines. This is called **resource reservation**. The resources reserved are used to construct an allocation set. This allocation set is offered to the job.

Question: So instead of waiting, the framework can ask for more resources?
Answer: There is no notion frameworks in case of Borg. A job can ask for more resources. A job itself could be framework, batch job or a web application. So Borg is not necessarily doing a two-level allocation.

There is ability to preempt. Lower priority jobs are terminated to allocate resources to a higher priority job.

Question: When you say terminate, are we talking about pausing a job or killing a job?
Answer: Termination is a policy decision. The important part of preemption is to reclaiming the servers. If the job being terminated is a web server, it doesn't make sense to pause it. But if is a long running batch job, it would make sense to pause it.

Question: Is the priority determined based on interactive and batch or can we have multiple priorities?
Answer: There is a clear distinction between interactive and batch. The idea can be extended to have multiple levels. Kubernetes allows you to make multiple priority levels.

The coordinator of Borg is replicated 5 times for fault tolerance. The copies are synchronized using Paxos(would be studied in greater detail in Distributed Consensus). It is designed with high degree of fault tolerance.

The scheduler has a priority queue. So the scheduler can either be best-fit or worst-fit depending on whether you want to spread the load or concentrate the load.

7.3 Virtualization

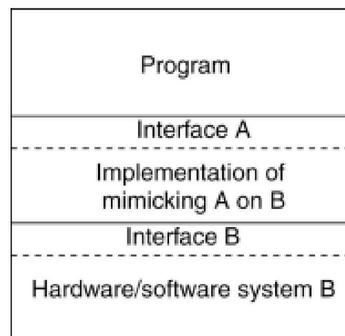


Figure 7.5: Virtualization

Virtualization uses or extends an existing interface to mimic the behaviour of another system. It is introduced in 1970s by IBM to run old software on newer mainframe hardware. It provided a software layer which emulates an old hardware, so an old software could be run on top of the software layer.

Question: Do mainframe machines have an operating system?
Answer: Yes, OS/360 and many generations of it. Early versions of UNIX came out from those OSes.

Question: What part of the image is the software layer?
Answer: The part between Interface A and B. That is the virtualization layer and it is implemented in software.

7.4 Type of Interfaces

Using virtualization, different layers of the hardware-software stack can be emulated.

- Assembly instructions (Hardware virtualization)

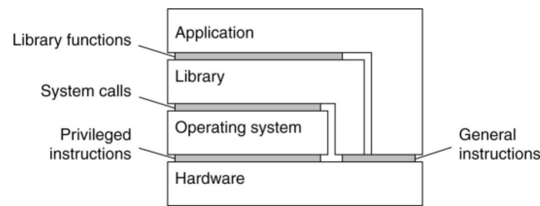


Figure 7.6: Types of interfaces in virtualization

- System calls (OS-level virtualization): E.g., The open-source software Wine emulates Windows on Linux.
- APIs (Application-level virtualization): E.g., JVM

7.4.1 Emulation

In emulation, a virtual machine (VM) fully emulates every aspects (CPU, a disk, a network interface card) of other machine. It is a software simulation of hardware. This is the most flexible technique as it can emulate any hardware on any hardware. But it's also the slowest as every machine instruction has to be translated and emulated.

Examples: Bochs, VirtualPC for Mac, QEMU.

7.4.2 Full/Native Virtualization

The VM does not emulate the entire machine, but it emulates enough hardware to run another system. This is typically done when an emulated interface and a native interface are the same.

Applications of nature virtualization include:

- Running a completely different OS on top of a host OS.
- Acting as a sandbox for testing.
- Running multiple smaller virtual servers on a single server.

Examples: IBM VM family, VMWare Workstation, Parallels, VirtualBox.

Lecture 8: February 22

Lecturer: Prashant Shenoy *Scribe:* ***Basundhara Chakrabarty***

8.1 Virtualization

Virtualization is the process of providing an interface to mimic the behavior of another system or component. Typically, virtualization mimics one of the following:

- Assembly instructions
- System calls
- APIs

Depending on what is replaced or mimicked, we obtain different forms of virtualization.

8.1.1 Types of Virtualization

8.1.1.1 Emulation

In emulation, a software simulation of a particular type of hardware or architecture is done using another. Once you've the emulated hardware, you can run an OS on it. An example would be to emulate an Intel process on ARM hardware, and thereafter any OS that was compiled for ARM will run unmodified on the emulated Intel processor. The OS will execute machine instructions on the emulated hardware, which won't run as-is on the native hardware, and therefore binary translation will need to be performed to convert them to native instructions. This caused significant overhead/slowdown, which causes a performance penalty.

Examples: Box, QEMU (Used by much Linux software), VirtualPC for MAC (Before, MAC computers were PowerPC based, and VirtualPC emulated a x86 software layer on PowerPC machine, allowing one to run Windows)

8.1.1.2 Full/Native Virtualization

In native virtualization, one or more unmodified OSs and the applications they contain are run on top of virtual hardware, given that the underlying native hardware and virtual hardware are of the same type. Here the VM simulates "enough" hardware to allow the OSs to be run in isolation. One limitation is that one can run OSs designed for the same hardware only. One use case is that the virtual OS can be used as a sandbox for testing components in different environments.

Examples: IBM VM family, VMWare Workstation, Parallels, VirtualBox.

8.1.1.3 Para-virtualization

Similar to full/native virtualization except that the kernel of the guest OS is modified so that it uses special APIs to call the hypervisor instead of directly accessing the hardware. The applications run are unmodified.
Examples: Xen, VMWare ESX Server.

8.1.1.4 OS-level Virtualization

Here the OS allows multiple secure VMs to be run on top of a native OS kernel. Each application run on a VM instances sees an isolated OS. This is lightweight as different OSs are not run.

Examples: Solaris Containers, BSD Jails, Linux Vserver, Linux containers, Docker.

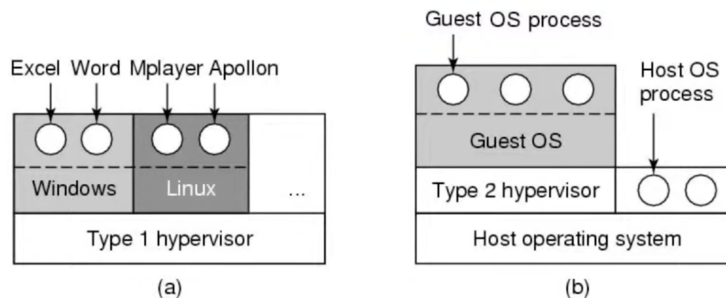
8.1.1.5 Application level virtualization

This type of virtualization uses an application interface to emulate another application. Here, the application is given its own copy of components that are not shared like global objects, registry files etc.

Examples: JVM written using C libraries exports the JAVA interface. Rosetta on Mac (also emulation) allows us to run binaries compiled on one architecture on another architecture WINE.

8.2 Hypervisors

The hardware virtualization layer is called a hypervisor or a virtual machine monitor (VMM). A hypervisor is the virtualization layer, which takes care of resource management, isolation and scheduling. There are 2 types of hypervisors that act like real hardware and implement full/native virtualization.



8.2.1 Type 1 Hypervisor

Type 1 hypervisor runs on ”bare metal”. It can be thought of as a special OS that can run only VMs as applications. On boot, it’s the hypervisor that boots.

Question: Do the multiple OSs running expect the same hardware?
Answer: Yes, because the hypervisor gives the illusion that the OS-s is running on the underlying native hardware. The hypervisor allocates resources between VMs, schedules VMs, etc., the hypervisor behaves like a pseudo-OS for the VMs.

A Type 2 hypervisor runs on a host OS and the guest OS runs inside the hypervisor.

Question: Does type 1 belong to native or paravirtualization?
Answer: Both.

8.2.2 Type 2 Hypervisor

The boot process boots the native OS, the hypervisor runs as a application on the native OS. The native OS is called host OS and the OSs that run on the hypervisor are called guest OSs.

Question: Can you do further virtualization?
Answer: Yes, but it is complicated. It is called nested virtualization

Question: Can multiple guest OSs be run?
Answer: Yes, multiple VMs can be run, each with its own guest OS

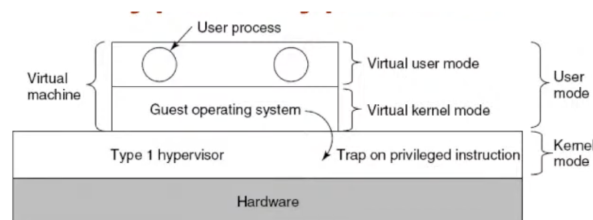
Question: Is the bootloader GRUB a hypervisor?
Answer: No, GRUB used in dual boot machines decides which one OS it has to boot, whereas hypervisors can run multiple guest VMs.

Question: Can a single hypervisor simulate multiple hardwares?
Answer: No, for that emulation needs to be used.

8.3 How Virtualization Works?

For architectures like Intel, different rings signify different levels of privilege. The CPU can run in either user mode (ring 3) or kernel mode (ring 0). In kernel mode, any instructions can be run. But in user mode, only a subset of instructions is allowed to run. The OS kernel is trusted more than user applications, so the OS kernel is run in kernel mode and user applications are run in user mode to limit instruction sets. A simple example is the halt instruction which user applications are not allowed to run. Also, memory management instructions are only allowed to run in kernel mode. Intel CPUs have intermediate rings (ring 1, ring 2) in which guest OSes can be run. Sensitive instructions (I/O, MMU, halt, etc.) are only allowed to run in kernel mode. Privileged instructions cause a trap or interrupt. I/O is one example. These are not related to user mode or kernel mode.

8.3.1 Type 1 Hypervisor

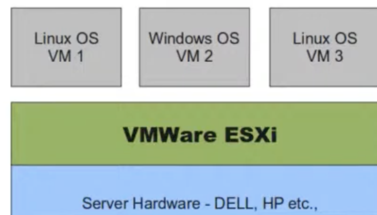


Theory: Type 1 virtualization is feasible if sensitive instruction is subset of privileged instructions or all

sensitive instructions always cause a trap.
Reasoning: On booting a Type 1 hypervisor, it runs in kernel mode. A Windows VM run on the hypervisor should not be trusted as much as the hypervisor and is therefore run in user Mode. Windows assumes it is the kernel and can run sensitive instructions, but these sensitive instructions won't run because it will be running in user mode. The solution is that the hypervisor intervenes and runs each sensitive instruction attempted by the Windows VM. How will the hypervisor be alerted when Windows attempts so? If the sensitive instruction causes a trap, the hypervisor intervenes and executes it for the VM.

Early Intel processors did not have Type 1 support. Recent Intel/AMD CPUs have hardware support, named Intel VT and AMD SVM. The idea is to create containers where a VM and guest can run and that hypervisor uses hardware bitmap to specify which instruction should trap, so that sensitive instruction in guest traps to hypervisor. This bitmap property can also be turned off.

Examples:



a) VMWare ESXI running, a specialized OS kernel that can run any arbritary VMs on it

b) Windows Hyper-V creates partitions, runs Windows in the parent partition (one copy of windows is mandatory), and the child partitions can run Linux or any other OS. Less flexible than ESXI.

c) Linux KVM or kernel virtual machine: Implemented as a device driver that gives barebone support for Type 1. Along with some other components (like QEMU) gives the functionality for Type 1 hypervisor.

8.3.2 Type 2 Hypervisor

A Type 2 hypervisor runs as an application on the host OS and therefore does not have kernel level privileges. Again, only the host OS can run in kernel mode. The Guest OS can also run only user mode instructions.

Therefore, the Type 2 hypervisor performs *dynamic code translation*. It scans instructions executed by the guest OS, replacing the sensitive instructions with function calls. These function calls in turn make system calls to the OS, thereby involving the host OS. This process is called binary translation. This leads to slowdown as every piece of sensitive code has to be translated. Therefore, VT support is not needed, no support is needed from the hardware to ensure that all sensitive instructions are privileged.

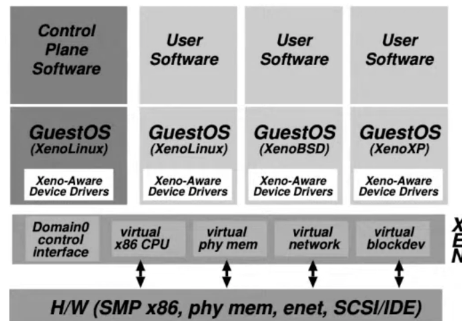
For example, VMWare Fusion, upon loading program, scans code for basic blocks and replaces sensitive instructions by VMWare procedure using binary translation. Only the guess OS's instructions need to be scanned, not the applications.

8.3.3 Para-virtualization

Instead of dynamically translating the sensitive instructions, leading to overhead, we can categorically change the kernel instructions to replace the sensitive instructions with hypercalls (call to the hypervisor) to get a

modified OS with no sensitive instructions. Every time a sensitive function needs to be executed, a hypercall is made instead.

Both Type 1 and 2 hypervisors work on unmodified OS. In contrast, para-virtualization modifies OS kernel to replace all sensitive instructions with hypercalls. Thus, the OS behaves like a user program making system calls and the hypervisor executes the privileged operation invoked by hypercall.



Example: Xen ran as a para-virtualized version of Linux when the hardware did not support Type 1 hypervisors. Xen needed Linux to run in domain 0 (master partition) just like HyperV.

8.3.4 Memory Virtualization

The hypervisor has control over RAM and allocates memory to the guest OS, and the guest OS allocates memory to its processes. If the guest OS tries to change the memory allocation of a process, that is a sensitive instruction because only the host OS is allowed to change page tables. Processes are not allowed to this. But a guest OS in Type 1 does not have those privileges. It still maintains page tables, and it still thinks it owns the machine that it can do whatever it wants, but it is not allowed to do so.

So what we do here is that we change those page tables in guest OSes to be read-only. When OS tries to write to that page table, a trap is created because that is a write instruction to read-only memory page. The hypervisor maintains a second page table which is called a shadow page table. That is a mirror of the original page table. It makes those changes in its actual page table. It utilizes the existing hardware feature that causes a trap when a write instruction happens on read-only region.

8.3.5 I/O Virtualization

Typically, the OS manages the physical disk, the guest OS should not be able to make changes to it. The hypervisor creates a large file on the guest's file system that emulates a disk called virtual disk (eg., vmdk for VMWare) When the guest OS writes to the disk, it effectively writes to this virtual disk. Multiple virtual disk maps to the real disk.

8.3.6 Benefits of Virtualization

One major benefit is that virtualization makes it easier to distribute pre-built applications and software. One can design virtual appliances (pre-built VM with pre-installed OS and pre-configured applications) that are plug-and-play.

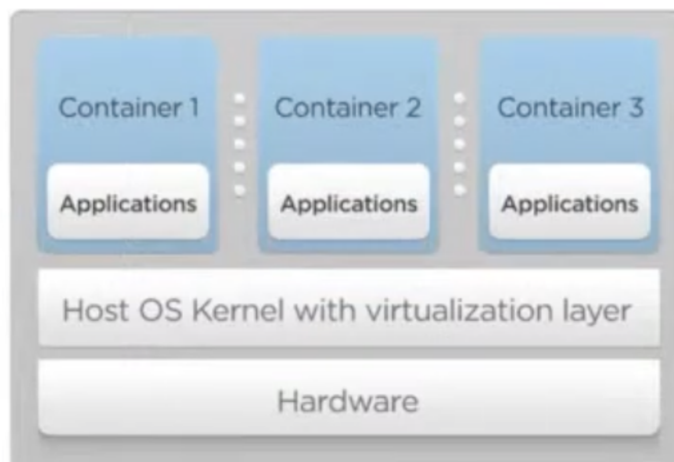
For multi-core CPUs, careful allocation of CPU resources per VM can be made.

8.3.7 Use of Virtualization

a) Cloud Computing b) Data Centres c) Software Testing and Development

8.3.8 Brief: OS Virtualization

OS virtualization uses the native OS interface to emulate another OS interface. A use case can be emulating an older version of OS. A popular use case is to allow backward compatibility, allow an older version of an application to be run, or sandboxing.



CMPSCI 677 Operating Systems **Spring 2022**

Lecture 9: February 23

Lecturer: Prashant Shenoy *Scribe: Samriddhi Raj (2022), Xingda Chen (2019)*

9.1 OS Level Virtualization

OS level virtualization is the use of the native OS interface to emulate another OS interface to mimic system calls. Lightweight virtual machines have no hypervisor. There exist containers that are light weight VMs which are constructed using OS level virtualization. This is a virtual machine that allows user to take resources from physical machine and construct containers. Each container can run one or more applications. All containers are managed through native operating systems. Difference between OS and hardware level virtualization including the following: hardware level virtualization has the abstraction of the hardware, which is emulated by the virtualization layer. You first have to run the OS in the VM and then the application runs in the OS. In OS level virtualization, when there are containers, the OS doesn't run in VMs. The OS is the same as what it runs on the actual machine, the containers runs on the native OS. Applications run using OS level virtualization are "sand-boxed", which means that don't see other running applications and they only see the resources allocated in their only containers. Containers are an isolated set of processes.

Benefits of putting applications into containers include:

1. : Isolation properties, better CPU usage allocation
2. : Security, limiting which files an application can see
3. : Good way to distribute software

Question: Why is creation and start-up of containers much faster than virtual machines ?
Answer: To start a container we just need to start an abstraction of operating system and put restrictions in them as to what the processes inside can access and see. Whereas to start a VM, we need to boot the entire virtual operating system. Thus, containers have faster provisioning due to less overhead during the OS boot.

Question: Does a malicious application in a container corrupt the host OS?
Answer: Yes, a malicious application can corrupt the host OS. In hardware level virtualization is this not the case? The OS is running inside the VM, if one VM is down the other VM continues to run. OS virtualization has less security than hardware virtualization? Every VM gets its own OS.

Question: In Docker how we can sometimes use different operating system like an image inside the contianer?
Answer: Docker uses linux containers. But the entire kernel is not booted up inside there. We can use application packages of a different OS distribution in there so that it looks like this corresponding OS environment. These application files exposed inside the container makes it look like a specific distribution. But there is only one underlying operating system being used, for example, Ubuntu.

Linux containers are lighter weight compared to hypervisors. Provisioning is fast due to a simpler structure. The container can emulation different OS interfaces. There is only one scheduler which resides in the OS. The scheduler in the OS ensures each container only gets its assigned CPU time.

9.2 OS Mechanisms for LXC

Container Abstraction is designed using two methods:

1. Namespace - Restrict what the applications/processes inside a container can see.
2. CGroups (Container Groups) - Restrict what the applications/processes can use. For example: CPU/RAM Usage (Resource Isolation).

Namespace and CGroups are independent abstractions that allow the implementation of the full container abstraction. In Linux both are designed as independent abstractions.

Additional Abstractions on top of baseline container abstractions include:

1. chroot: changes the root directory to a user specified directory.
2. Docker offers SDE tools that allow to create and distribute images as containers.

9.2.1 Namespaces

Namespaces restrict what a container can see. Some of namespace restrictions are:

1. Which other processes may be seen. For example, 3 Processes inside a container can see each other. They cannot see other processes running on other containers or on underlying OS.
2. Abstract Process IDs of other processes.
3. Restrict File Systems/ Directories/ Mount Points - Storage Restrictions
4. Network Detail Restrictions
5. User ID Visibility

Example: A machine has total 10 user accounts. Only 2 user accounts are visible on a container. The other 8 users cannot access this container.

9.2.2 Linux Cgroups

CGroups puts an upper limit to the amount of resources a container can use. Example - A machine has 16GB RAM and 4 cores, but the container can also use 1GB RAM and 1 core respectively. Specialized scheduler like weighted proportional share scheduler can be used to allocate limited resources to a container.It allocates a weight to a container and CPU time is allocated to the containers in proportion to the weight. In the situation when the container is not utilizing the assigned CPU time, hard allocation will just waste the CPU time and fair resource distribution will redistribution the cycle accordingly, it is also called work concerning scheduler. A different technique known as Core Pinning can be used to assign container to a particular core. CPU Scheduler/OS will not run processes belonging to this container on other cores.

9.3 Proportional Share Scheduling

This mechanism is widely used in virtualization to allocate resources to individual virtual machines (Type 1 and Type 2 Hypervisors) and containers. It is used to decide how much CPU weightage to allocate to each container and network bandwidth to each container. In share-based scheduling, a weight is allocated to each container and CPU time is divided in proportion to this weight. So if two containers have 1 and 2 as weights they will receive 1/3 and 2/3 of the CPU time.

Hard limits: Hard limit means there is a hard allocation,no matter if a process uses it or not, it is allocated.

9.3.1 Weighted Fair Queuing (WFQ)

Each container is assigned a weight w_i and it receives $w_i / \sum_j w_j$ fraction of CPU time. The scheduler keeps a counter for each container, s_i, which tracks how much CPU time each counter has received so far. The scheduler picks the container with minimum count so far and allocates a quantum time unit q. The counter value for the selected process is updated, $s_i = s_i + \frac{q}{w_i}$.

If one container is blocked on I/O and not using CPU, its counter value does not increase. However another container keeps on running and its counter value is incrementing. When the first container finishes I/O, it will have a low counter value as compared to second counter. Thus it will be scheduled for a long time while the second container is starved. This does not follow fair based scheduling. To avoid this the counter value is updated using this formula: $s_{min} = \min(s_1, s_2, \ldots)$ and $s_i = \max(s_{min}, s_i + \frac{q}{w_i})$.

9.4 Docker and Linux Containers

Docker uses abstraction of Linux containers and additional tools for easy management.

1. Portable Containers - With LXC, we would have to use namespaces and cgroup commands to construct a container on a machine. With Docker, all of this information can be saved in the container image and this image can be downloaded and run on a different machine.
2. Application Centric - Docker can be used for designing applications quickly. Software can be distributed through containers.
3. Automatic Builds
4. Component Reuse - Helps to create efficient images of containers by only including libraries/files not present in underlying OS. Achieved using UnionFS.

Question: Does Docker deal with packages dependencies for the applications running in the container?
Answer: Packages needed for an application are included in the Docker image. Specific version libraries that are required for application need to be included in the image if they are not present in native OS. OS Does not need to worry about these libraries or issue of incompatible versions.

Question: Docker images are also available for Windows/MAC. How is it possible to run Linux Containers on Windows/MAC ?
Answer: Docker provides a hidden VM of linux and containers run on top of it. These are small barebone linux kernels that run on non-linux platform. Docker does not use true OS virtualization for running linux container on other platforms as it would have to translate linux calls to other platform calls (Windows/MAC Calls).

PlanetLab: Virtualized architecture used for research by students in different locations.

9.5 Code, Process and VM Migration

The motivation behind developing techniques for code, process and VM migration is that migrating these components of a system helps improve performance and flexibility.

There are two types of migration models:

- **Process Migration:** Also known as strong mobility, this includes the migration of all the components of a process, i.e., code segments, resource segments and execution segments. An active process (an already executing program) on a machine is suspended, its resources like memory contents and register contents are migrated over to the new machine and then the process execution is restarted. It involves significant amount of data transfer over the network.
- **Code Migration:** Also known as weak mobility. In this model only the code is migrated and the process is restarted from the initial state on the destination machine. The network transfer overhead is low since only the code is transferred. Some examples of code migration are web-form, Flash/Java applets in browser, web search. Docker is also considered to be an example of code migration.

Question: Why is a search query an example of code migration?
Answer: Keywords typed in a search bar basically become part of a query and query is a program. On pressing submit, the query is sent to another machine and is executed there. This illustrates migration of code from client machine to the server machine.

Question: In process migration, if you suspended an active process and migrated it, how do you take care of its state?
Answer: The specific state of process like its memory contents can simply be written onto a disk and the process can be resumed elsewhere. Debuggers perform a similar operation.

Question: Does the migration have to be only between client and server or can it be between a cluster of servers?
Answer: Migration is not limited to just client and server. Migration is independent of the source and destination of the code or process.

An example of sender-initiated migration is a web search and database query. An example of receiver-initiated migration is a browser downloading a Java applet or Flash application from the server.

A process can be migrated or cloned. In case of migration, the complete process is moved to a different machine. In cloning, a copy of the process is created on a different machine and you allow both the copies to execute. Cloning is a convenient way of replicating the process. An example of cloning is forking a process.

To decide whether to migrate a resource attached to a process or not, we look at the nature of binding of resource to process. There are three types of resource to process bindings:

- **Identifier:** Hard binding, one that you cannot substitute. Least flexibility. Example is URL for a website
- **Value:** Slightly weaker binding. Libraries used in Java are a good example.

- **Type:** Weakest binding, one that can be substituted. Maximum flexibility. Example is a local device like printer.

Apart from the resource binding, it is also necessary to look at the cost of moving resources which can also be classified into three categories:

- **Unattached:** Very low cost of moving. Example- files.
- **Fastened:** High low cost of moving. Example- databases.
- **Fixed:** Can't be moved. Example- local devices.

Different combinations of resource-to-machine binding and process-to-resource binding are tabulated below:

	Unattached	**Fastened**	**Fixed**
By Identifier	MV (or GR)	GR (or MV)	GR
By Value	CP (or MV, GR)	GR (or CP)	GR
By Type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

where GR means establishing global system-wide references, MV means moving the resources, CP means copying the resource and RB means rebinding process to locally available resource.

Question: Would you want to do checkpoint and restart instead of migrating a process?
Answer: Checkpoint and restart is a standard way to implement process-migration and not an alternative to process-migration.

Question: Would you want to do checkpoint and restart instead of migrating a process?
Answer: Checkpoint and restart is a standard way to implement process-migration and not an alternative to process-migration.

Question: If JVM is a value resource, why can't we bind JVM of the new machine to newly migrated process ?
Answer: If exact same version of JVM as required by the process is available on the new machine, then we can bind it to the process. However a different JVM version can be incompatible and cause the program to crash.

CMPSCI 677 Distributed and Operating Systems **Spring 2022**

Lecture 10: Feb 28

Lecturer: Prashant Shenoy *Scribe:* ***Shiksha Rawat, Bharath Narasimhan***

10.1 Virtual Machine Migrations

Process and code migration in heterogeneous systems, i.e., systems in which the destination architecture is different from source architecture, pose a challenge. Migration via interpreted code is one possibility but it is often clumsy in practice and is almost never used. Moreover, interpreted code migration only supports weak mobility. In such cases, it is beneficial to look at techniques for VM migration.

VMs can be migrated from one machine to another irrespective of architectural differences. A VM consists of its OS and some applications running on this OS. So, in VM migration the OS and these applications are migrated with negligible down time. As the processes inside a VM will also move, VM migration also involves process migration. VM migration is usually done live, that is, it keeps executing during migration. Applications continue to run, nothing has gone down, and then after a while the VM disappears from one machine and shows up on another machine with the applications still continuing to run.

There are two methods for VM migration:

Pre-copy Migration: The process of pre-copy migration involves the following steps:

1. Enable dirty page tracking. This is required to keep a track of pages which have been written to.
2. Copy all memory pages to destination.
3. Copy memory pages which were changed during the previous copy.
4. Repeat step 2 until the number of memory pages is small.
5. Stop VM, copy rest of memory pages at destination and start VM at the destination.
6. Send ARP packet to switch

Post-copy Migration: This is also called lazy copy. The process of post-copy migration involves the following steps:

1. Stop VM and move non-memory VM states to destination
2. Start executing on new machine
3. In case of page faults in the new VM, copy the page from the source machine. Copying of other pages is also started in the background. Background copying also ensures that every page is copied even if it was required during a page fault before the VM is deleted from source.

One advantage of post-copy over pre-copy is that there is no iterative copying and each page has to be fetched only once. Pre-copying is preferred in some cases when we do not want applications to keep waiting in case of page faults so there is less impact on application performance.

Question: How much copying is enough to restart VM at the destination?
Answer: Moving at least all the registers, a few OS pages that program counter is pointing to, should be good enough to restart the VM. More details depend on the hardware architecture.

Question: Programs often have spatial and temporal locality of references, can we use it to intelligently figure out what to pre-fetch?
Answer: Post-copy migration can make use of such optimizations where for example, one can get the working set of the programs first and then fetch the rest.

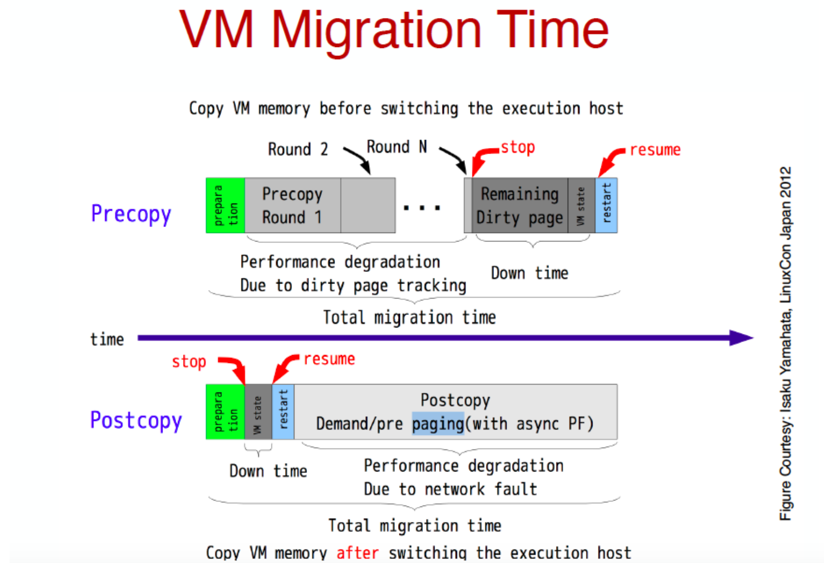


Figure 10.1: Visualization and comparison of pre-copy and post-copy VM Migration.

10.2 Types of Migration

There are three types of migration:

Cold Migration

1. VM is not running. There is just an image of VM/container or virtual image of VM/container on the disk.
2. Copy this image and data files to new machine.
3. Start on new machine.
4. No state preserved.

Warm Migration

Incurs downtime to migrate state from previous instance, but preserves the state.

1. Suspend running VM/container to disk.
2. Copy image, data, suspended memory state.
3. Resume execution of suspended VM.

Hot/Live Migration

Migrate state but with no downtime so copy state while VM executes. Most complex to execute.

10.3 Container Migration

Containers are light weight VMs. When you are migrating a container you are migrating only the processes and some resources that it accesses. The underlying OS is not getting migrated.

10.4 Snapshots

A snapshot is a copy of some object (file, disk, VM, container) at a certain point in time (point-in-time copy). We can preserve the contents of the VM(the memory and disk content) as they existed at that point, if one takes a snapshot of a VM at a certain instant. Snapshots are also known as checkpoints. Snapshots help in rolling back to a point in time and creating backups.

There are two ways to create a snapshot.

1. Full (Real) Snapshots - Actually make a real copy. Very inefficient.
2. Virtual Snapshots - Here a virtual copy is created. Instead of making a real second copy of a file we just copied the metadata and all of that is pointing to the previous copy. The previous copy can continue to change because the VM can write on the disk. So, in case of virtual snapshots, copy-on-write is used. Whenever a VM is about to write to a previous copy, a new copy is created first and pointers are shifted to this new copy. Virtual snapshots are very efficient as compared to full snapshots.

Question: Snapshots are useful for migration. Is it only valid for real snapshot real copies as opposed to a virtual copy?
Answer: Even in a virtual copy you have access to all of the data because it's just pointing to the same blocks. So, it will not matter whether you took a virtual snapshot or a real copy for migration purposes.

10.5 CheckPoint and Restore

This is a warm container migration technique. Many containers actually support checkpoint and restore as the first option because live migration is a bit more complicated. Migration in containers is a little more complicated than VMs as in containers we only migrate processes. Steps of CheckPoint and Restore are :

1. Pause container execution.
2. Checkpoint (save) memory contents of container to disk.
3. Copy checkpoint to new machine (memory + disk image).
4. Resume execution on new machine.

Question: Is a checkpoint the same as a snapshot?
Answer: Checkpoints are like real snapshots.

10.6 Linux CRIU

1. Linux CRIU (Checkpoint Restore In User Space) : It is used for warm or live migrations, snapshots and debugging. CRIU is not a container-specific technique, it is a process-specific technique. As a container is a collection of processes, it suspends all of the processes in the container and writes it out on the disk one by one.

2. CRIU uses `/proc` file system to gather all info about each process in the container. In the proc file linux keeps its OS data structures. It's like a file system—you can go and look at the files. There's information about every active process in the system.

3. CRIU copies saved state to another machine.

4. CRIU restorer

   (a) Use fork to recreate processes to be restored

   (b) Restorer also restores the resources being used by processes; for container, restores namespace

   (c) If any network connections were being used, we have to do extra work if we have migrated the container to a new machine. We can migrate active sockets only if the IP address moves along with the container to the new machine. To do so, we can use virtual network devices in containers and move them.

Question: Why can't the process bring the IP address of the previous machine?
Answer: We can not do this because the socket connection is always tied to the IP address on which socket was established.

Question: If we assign a new IP address to the container, how to make sure it is unique?
Answer: We can have either a public IP or a private IP. If it's a public IP by definition it has to be unique. Many containers will not have a second public IP. They'll give themselves a 192 address which is a private IP address. So when you move a container over a new machine, to make this private IP work, the new machine should not have the same IP address running on it already.

10.7 Kubernetes (k8s)

Container orchestration is a form of cluster scheduling but rather than scheduling jobs or http requests you're scheduling containers. There is a pool of machines and applications are coming as containers. The goal of the container manager is to now assign this container onto some physical host. When the application is done, the container is terminated. The scheduler does not care about what is inside the container, it just schedules the container on the basis of its resources requirements.

Kubernetes is one of the most popular container orchestration systems. It is based on Google's Borg/Omega cluster managers. In Kubernetes, it is assumed that all applications are containerized. K8s will deploy them onto machines of the cluster. Kubernetes provides the following features:

1. Replication of apps on multiple machines if requested (fault tolerance)

2. Load balance across replicas

3. Can scale up or down dynamically (vary replica pool size, a concept similar to dynamic thread/process pools)

4. Provide automated restart upon detecting failure (self-healing)

10.8 K8s Pods

Kubernetes has the concept of pods. A *pod* is an abstraction where we can have more than one container in it. Containers inside pods share volumes and namespace. Kubernetes doesn't directly deal with containers, it deals with pods. So, pods are the smallest granularity of allocation in k8s.

In a distributed application, because your application has multiple components in the kubernetes world each component has to be containerized first. So, each pod consists of one or more components/containers.

Pod can contain all containers of an application but if a component needs to be scaled, put that component in a separate pod. As a good design principle, each independently scalable component should be put in a different pod. Constructing applications in pod is the job of an application developer. Deploying and scaling it is the responsibility of kubernetes. Pods of an application can span multiple cluster machines.

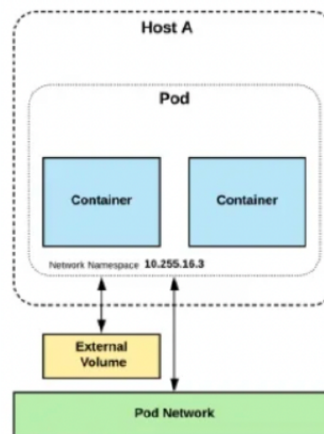


Figure 10.2: Visualization of the relationship between containers, pods, and hosts.

10.9 K8s Services

Pods are used to construct kubernetes services. A service is a method to access a pod's exposed interfaces. Features of services include:

1. Static cluster IP address
2. Static DNS name
3. Services are not ephemeral
4. Collection of pods

Pods are ephemeral. Each has its own IP. They can be migrated to another machine. Pods can communicate with one another using IP.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 11: March 4

Lecturer: Prashant Shenoy *Scribe: **Nikhil Agarwal***

11.1 Kubernetes (k8s)

Kubernetes is a cluster management system using containers as its fundamental abstraction. It is based on Google's Borg and Omega cluster schedulers.

Kubernetes uses another abstraction called 'pod.' A pod is an abstraction which contains on or more containers. Kubernetes deals with resources and allocate resources to pods. Kubernetes takes pods, put them on machines, and then allocate the resources of that machine to the pods. This means that Kubernetes doesn't deal with containers directly but instead deals with pods. However, we can put just one container inside a pod and then a pod is essentially very similar to container. How different components are put inside pods is a design choice for the application. A simple approach could be to put every component (like a database or a web server) and put it inside its own pod. We can replicate pods as well. For example, to run three instances of a web server, we can put it inside a pod and then replicate this pod a total of 3 times.

11.1.1 Kubernetes services

A Kubernetes service is how Kubernetes exposes pods to the outside world by giving them a public IP addresses through which clients like web browsers can interact with them

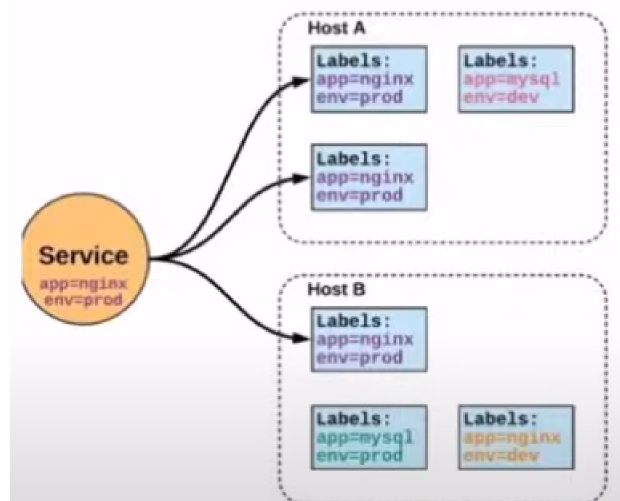


Figure 11.1: A Kubernetes service exposing a web server.

In Figure 11.1, the blue boxes are the pods (the containers are not shown in this figure) which is a replicated server exposed to the outside world through this service using a single IP address. In addition to exposing

the web server, the service also performs functionalities like load balancing. Note that a service is a durable entity, i.e., it is kept around by Kubernetes until we decide we no longer need the service. A pod is a much more temporary abstraction, an ephemeral entity. This means that Kubernetes can terminate or migrate pods if needed. Also, the mapping of pods to machines can vary over time and the number of pods k8s creates can vary over time. A service however remains durable even if the pods it exposes are changed over time.

11.1.2 Kubernetes Control Plane

Control plane is the controller of the Kubernetes which manages the machines in the cluster. Here is a picture of different components of the control plane.

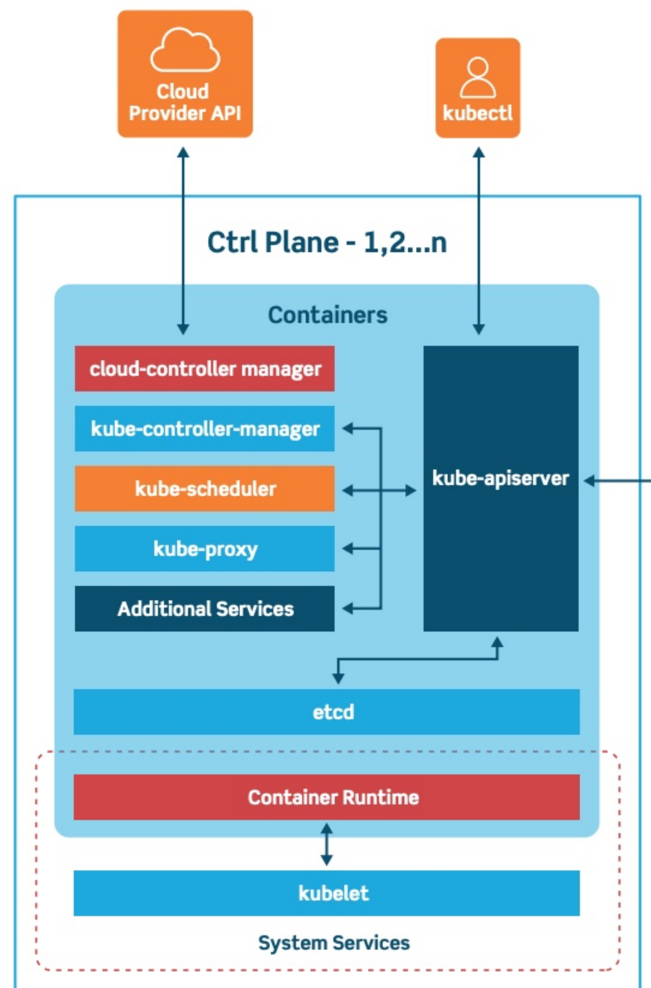


Figure 11.2: A Kubernetes control plane

The components of the control plane are as follows:

- **apiserver**: Exposes REST interfaces to the users using which they can pragmatically interact with the k8s controller for management, like starting a new application or requesting more resources. Please note that this is the controller API and not an application API.
- **etcd**: A key-value datastore which is kept on the disc because it stores the state of the k8s which needs to be persistent. It stores information like which pods are mapped to which machines, running status of the applications etc. It provides properties like high durability (RAFT consensus) and strong consistency. Replication of datastore can be done onto multiple nodes for fault tolerance consistently using something known as RAFT consensus.
- **controller-manager**: Used to create and replicate pods, monitor health of the pods and the nodes on which they are running, and to restart pods. For example, we can specify a replication degree of four and the controller-manager will replicate the pod on four machines and restart it on another machine in case of failure to maintain the replication degree. Note that this is done without developer intervention. Controller-manager ensure that failure recovery is built-in the system. This is not common in other cluster schedulers.
- **scheduler**: Scheduler is the entity that assigns pods to servers based on resource constraints and policies like choosing least loaded machine.
- **cloud-controller-manager**: This entity only comes into play if the Kubernetes is running on cloud machines. The cloud-controller-manager interacts with the interface of the cloud platform to manage the Kubernetes. s

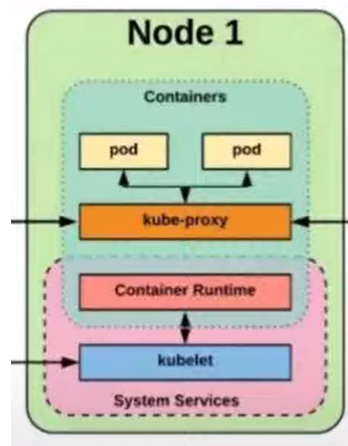


Figure 11.3: A Kubernetes node

Another important entity in the Kubernetes is the **k8s node** which is the actual machine on which the pods are running. There are several components within these nodes which are explained below:

- **kubelet**: On each of these nodes, a kubelet is running which monitors the health of the containers which are running on this machine and esures they are healthy. In case it detects a problem, it communicates with the control plane to take appropriate action about this incident.
- **kubelet proxy**: Kubelet proxy is the network manager which directs the request to the pods referenced over the network. This component also provides services like load balancing for incoming requests to cluster services.

- **container runtime**: The container runtime is the OS virtualization that is used by Kubernetes to run the containers. Kubernetes can use different kind of OS level virtualization as runtime like docker, cri-o etc.

Question: If pod is an abstraction which consists of containers, what do we mean when we say a pod has failed?

Answer: A pod fails when a container within it fails, which in turn fail if a process that runs within the container fails. Remember, pods and container are both abstractions and process is the entity which can fail. We can however, just take action on failed containers inside a pod and don't have to restart the whole pod. In the default case when a pod only has one container, if the container fails then the entire pod fails.

Question: Do the pods have their own IP address or the containers?

Answer: A pod has one IP address which is also the IP address of all the containers inside it. However, the containers host services at different ports at this IP address.

Question: Do the containers within a pod share the same namespace.

Answer: Yes, the namespaces are shared even though the resources are isolated within a container.

Question: How can the resources be isolated if the namespace is shared?

Answer: Namespace is what you can see and resources are what you can use. CPU, memory and other resources can be allocated at the container level.

11.2 Data Centers

A data center is a large collection of servers and storage that provides these computing and storage resources to applications. They are also referred to as servers and storage farms. They can be large or small depending on the number of servers (or racks of servers) or storage units that they have. All companies like Google, Meta, Amazon etc. are running their applications on servers which are sitting in a data center somewhere. These servers can be owned by these companies or by a different cloud provider. A company can create its own data center and use them or use the servers provided by cloud provider. Regardless of that, a data center can be used for variety of purposes like data processing, hosting web sites, hosting business apps, among many other functionalities.

11.2.0.1 Inside data center

What we see in Figure 11.4 is an aisle of a data center. We can see racks of data servers which collectively form this aisle. One data center has many such aisles of racks of servers. Each of these servers provide compute and/or storage resources. To oversimplify, we can say it is like a supermarket where we have servers on the shelves instead of products. Because these servers generate a lot of heat, a lot of cooling infrastructure is also needed. In addition to these, power generators/converters/backups are also needed for the servers and the coolers.

11.2.1 Data center architectures

Data-center architectures have seen a shift from traditional architecture to a modern day architecture. These are discussed below:

Figure 11.4: An aisle of a data center

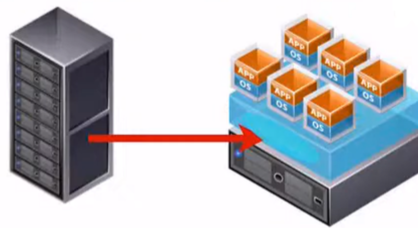


Figure 11.5: A virtual server

- **Traditional architecture**: A traditional architecture has applications running on physical servers. System administrators manage and monitor these servers manually. This architecture Storage Arrays Networks (SAN) or Network Attached Storage (NAS) to hold data. As a company or an individual in need of servers, you go and buy these servers from vendors, put them in a physical warehouse, and then run your application on these servers using a lot of manual configuration and decision making like which IP addresses are assigned to which servers and which applications run on which of these IP addresses.

- **Modern architectures**: Modern architectures automate a lot of the things in the traditional architecture. Instead of running applications on physical servers, they are ran on containers which are present on virtual machines mapped to physical systems. All the mapping from containers to virtual machines, and virtual machines to physical servers are done by an automatic controller instead of system administrator. For example, cluster managers like Kubernetes can be used for this purpose. Overloading can be managed using techniques like live migration. All these properties makes a modern architecture much more flexible and scalable.

11.2.2 Virtualization in Data Centers

In a **virtual server**, a hypervisor is layered over a server which allows different virtual machines or containers to be laid on top of it. These virtual machines can run different operating systems in them and these OS can run different applications inside them. An array of such servers is put on a rack in the data center.

The concept of **virtual desktops** is also becoming common these days where the actual desktops are placed on the virtualized data centers as explained above and the thin remote clients (with less resources) can

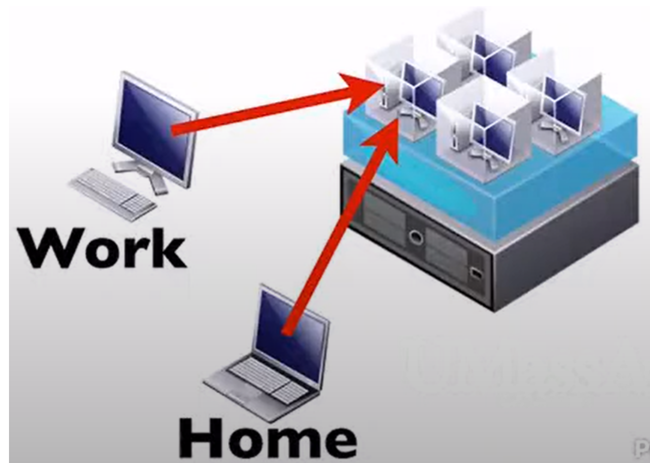


Figure 11.6: A virtual desktop

connect to these desktops over a graphical interface and access them from anywhere. This architecture makes the actual desktop physically separated from the end users. These virtual desktops are generally used by workplaces for their employees.

11.2.3 Server virtualization

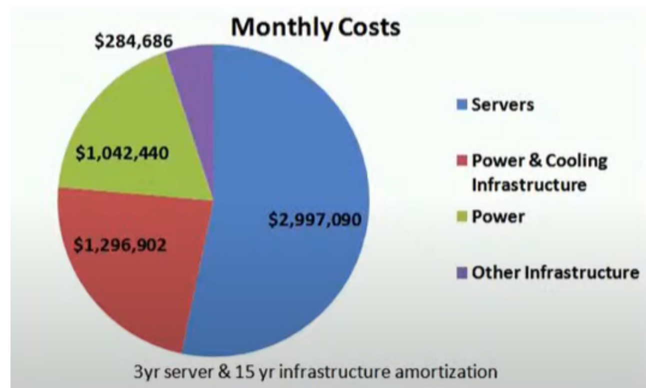
Server virtualization is very similar to virtualization of any other machine. A virtualization layer like hypervisor 1 or hypervisor 2 can be used. This allows for a server to be sliced into different virtual machine. This is done primarily because applications require only a subset of resources available. Virtual machines can be created based on different resource needs and the application can then run on these VMs. All benefits discussed like rapid resource allocation, fast live migration are available here as well. For example if there are two VMs on server VM1 and VM2, VM2 can be live migrated to another machine if VM1 needs more resources. These functionalities are not available on a physical server. This is the main reason most data centers and cloud providers use a virtualization architecture.

Question: Does the term virtual private server refer to virtualization?

Answer: It is indeed a virtual server which allows private access only to the person/entity it is assigned to.

11.2.4 Data center costs

Running a data center is expensive. In addition to powering up the actual servers present, additional costs also need to be paid like cooling infrastructure. There is a lot of power consumption in the form of electricity in the data center. A lot of work has gone into reducing the power foot print of the data center which could be mesaure by PUE (Power Usage Effectiveness). This is calculated as Total power/IT power as we want IT power to be the major chunk of the total power. IT power is the power used for servers. Ideally this should be 1 as we would want no overhead. This used to be around 2, which means there was a 100% overhead on every unit of power consumed by the server. Nowadays, typically this metric is around 1.7 but google has managed to take it down to 1.1 in its own data centers.



11.3 Cloud computing

Cloud computing is the use of remote servers to run distributed applications. Mostly the servers are running remotely on the cloud and clients are the users of the distributed application. The actual use-case of cloud computing can be much more detailed though. The biggest revolution in the field has been cloud computing platforms. The owners of the applications have been separated out from the owners of the cloud servers. The resources available on the cloud servers is provided to others (individuals or companies) as a service. The users can lease the remote resources from the data center as a service and pay the provider according to a planned policy. This decouples the applications development from the application deployment and IT infrastructure. There are many benefits of such a service like remote availability on the internet, high scalability (procure more resources on demand), high flexibility (pay only for shorter duration of usage times). In addition to these individual benefits, this service also creates a shared infrastructure which leads to a more economic use of total available resources. The focus has shifted from buying resources to leasing resources which has made all these benefits available to almost everyone.

11.3.1 The cloud stack

The services being operated on the cloud can be made available to the users in varying degrees of freedom and specificity. The lower stack of the cloud services we get, the more barebone hardware resources we get and the more we are in control of what and how to do things like deployment. The more up we get into the stack, the more things are already done and the lesser is our control on the allocation and use of the resources.

- **Infrastructure as a service (IaaS)**: In this the users get machines from the service providers. These can be the servers and/or storage units. What and how to do with these machines is decided by the user of the service. The metric for billing here is the scale and duration for which the resources are procured.
- **Platform as a service (PaaS)**: Not everyone will know how many resources/machines they need for their use-case. Platforms are better for such use-cases because the user just provide their app as a container and how to deploy it including decisions like scale of resources to use is decided by the platform provider. The metric for billing here is usually the number of requests served by the application.
- **Software as a service (SaaS)**: Neither the development, nor the deployment of the application service is done by the user. The user just has user level access to a software/application that is provided by the SaaS provider. Examples: GMail, google drive etc.

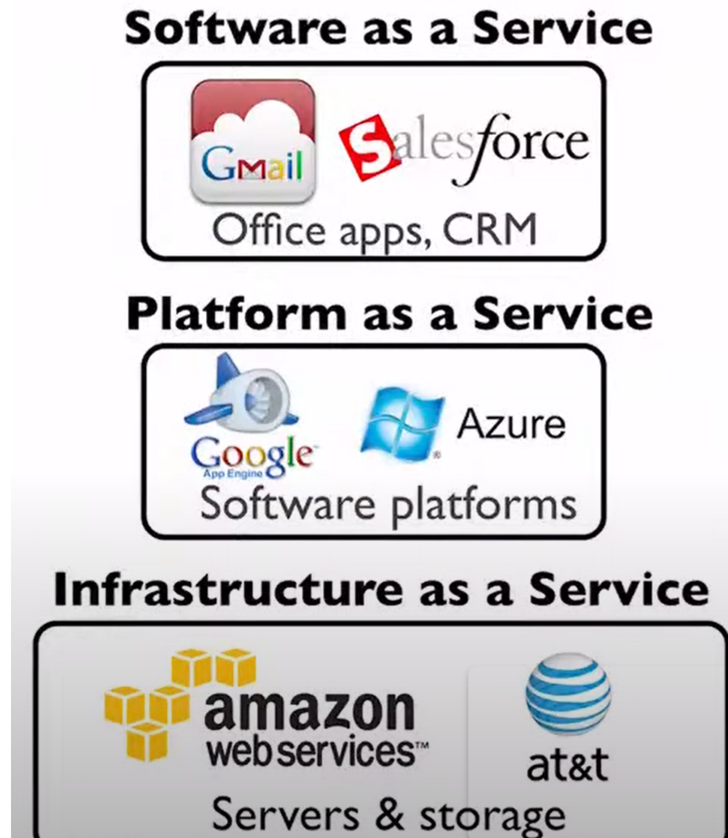


Figure 11.7: Cloud stack

11.3.2 IaaS: Amazon EC2

We are explaining IaaS by using the example of Amazon EC2 here. Remote servers and storage units are provided to the users as bare-bones. Billing is done for both servers and storage based on the configuration selected by the user. These range from smallest, medium, largest to other specific categories for both the servers and the storage units. The metrics for billing is generally CPU cores, memory (RAM), storage being provided. A sample billing policy has been mentioned below which has prices that can give a good idea about the actual billing. Note that all the resources are virtualized, and physical servers and disc are not allocated to the users. This allows for single servers and discs to be shared among multiple users.

11.3.3 Types of IaaS instances

In addition to different scale of the resources, the billing policy is also categorized on the basis of what is given as an abstraction in the service. These are described below:

- **On demand instances**: This is the most exclusive and expensive service provided. Here resources are requested and provided on the fly on demand. They are to kept with the user exclusively till the user decided to terminate this procurement. The billing is done by the minute and is generally the highest because of the services are provided exclusively and on a short notice. This model is generally

	Smallest	Medium	Largest
VCPUs	1	5	33.5
RAM	613MB	1.7GB	68.4GB
Price	$0.02/hr	$0.17/hr	$2.10/hr

Storage	$0.10/GB per month
Bandwidth	$0.10 per GB

Figure 11.8: Billing policy

used by larger companies/firms for unforeseen required resources.

- **Reserved instances**: Reserved instances are procured for longer duration of time (like 1 year to 3 years) and for cheaper prices compared to the on demand model. This is so because the provider can bills user for all the time and has his resource busy during this long duration. This model is preferred for most of the work done by larger firms/companies.

- **Spot instances**: These are instances of the resources that are provided at large discounts because these contain resources which have not been procured as part of the first two models and are currently free. However, these resources can be revoked after a warning within a short period of time if needed by some user in the on demand model. This model can be used during times where low load of processing is required. (off-peak periods). Because of the sporadic availability in this billing model, it is generally used for lower priority tasks and background tasks like batch jobs, large data processing, mapreduce and spark jobs. This model is not used for hosting real time applications which impact end users like websites. To use spot instances, jobs generally have systems like checkpoints built in them so that progress can be saved before a restart happens.

Question: Do cloud providers switch off the servers during off peak periods?

Answer: Yes, this needs to be done to save electricity which the provider itself has to pay for. They don't necessarily need to be shut down and can be put in a low power mode where they can be started quickly.

11.3.4 PaaS Cloud

In platform as a service cloud, instead of providing bare-bones server and storage units, the provider provides a platform on which applications are deployed. The biggest difference with IaaS is that the development of the application is separated out from the deployment of the application. The application developers just build their applications in frameworks like .NET, Java, Python, Node etc. and it is the responsibility of the platform provider to take care of deployment and scalability of the application. Since hardware is not provided as a granularity here, the billing is done on the basis of the usage of bandwidth of the provider which is directly related to the rate of requests served by the application. This model is also getting popular in cloud providers (serverless computing) where servers are not directly provided to the users. User provide their application container to the cloud provider which is hosted by them just as in PaaS.

11.3.5 Public, private, hybrid cloud

The cloud service which we have been discussing so far is known as public cloud which is available to the general public if they create an account with the provider and pay for it. This is called as the public cloud. However, many a times, companies are not comfortable with this kind of service as they don't want to share their disc time and servers with others. In this case, dedicated data centers or smaller parts of existing centers can be allocated for the use of just one company. All the benefits and services of virtualization are still available but the use is limited to a single entity to which the resources are dedicated. However, this might come at a higher cost because of the promised exclusivity.
A hybrid of these models is called the hybrid cloud in which the provider moves resources between the public pool and the private pool of servers. From the users perspective, this is done because their usage may have changed over time and they might want to reduce or increase the resources allotted to them. Accordingly, resources are moved to or from the public cloud. From the providers perspective, this is done for the more efficient use of resources so that they are kept busy as much as possible. There is also a financial incentive to move resources from public pool to hybrid pool. This usually happens when the clouds bursts, which means the user needs and starts using cloud servers and storage from what they had been allotted earlier.

Question: Why even a provision for something like a hybrid cloud?

Answer: Hybrid cloud is needed because the workload is dynamic and can be different from estimates because of daily and seasonal variations. Hybrid cloud provides a good workaround by moving resources on a need basis.

11.3.5.1 Cloud workloads

Cloud services can be used for a variety of purposes. Some of them have been discussed below:

- Client server: Cloud services can be used to host web servers and databases.
- Batch processing: Cloud services can be used for processing data as batches as need for business apps and payroll management.
- Data processing and analytics: Cloud services can be used for data intensive computing as needed for MapReduce and Spark.
- AI and Machine Learning: Recently, clouds have also been used for training machine learning model in which case servers with GPUs are being used.
- Specialized instances: Many of the highly specialized instances are conducted on supercomputers. Certain cloud providers now provide resources that can be used by such users which can be cheaper for them on a lease basis.

11.3.6 Cloud storage

Just like servers can be leased on the cloud platforms, storage can also be leased on the cloud platforms. Cloud platforms can give storage in the form of object storage in which any form of object like a file, a dataset etc. can be stored and retrieved using get and put methods. This storage is different from typical file systems that we deal with on most of the computers. Here, blobs of storage are used as the unit of storage. However, cloud platforms also have the provision for traditional disc storage/block storage on which a regular file system can be run. Note that in block storage, the disc is attached to a machine and only that particular machine can access that disc. A spin-off of this storage is the network file storage which is essentially a

networked file system where the disc storage is shared among multiple machines sharing a network. There is also a provision for archival storage which is slower and cheaper in nature. It is mainly used for storing backups of data which are used infrequently. One notable use of cloud platforms as storage is Dropbox which doesn't provide any servers to the users and only provides them space. The user can simply upload their files/data in the Dropbox folder in one machine and it is automatically synchronized and made available on the other machines running the same Dropbox account using cloud as the intermediary where the files are uploaded. Other popular storage over cloud platforms include Google drive, One drive, Box etc. which essentially gives the user space on cloud accessible over the internet. Other cloud storage models are also available like cloud backups (iCloud), cloud media storage (Google photos) etc.

11.3.7 Cloud orchestration

Cloud orchestration refers to the management of the cloud service. This is typically done via the cloud controller which is conceptually no different from a Kubernetes controller. Just like a k8s controller, it manages multiple customer requests using virtualization. VMs are created/migrated/terminated on servers to allocate and deallocate appropriate resources (like CPU, disc, network bandwidth) on the basis of the specific needs of the customer requests and the moving status of these requests. VMs can also be booted on the servers using specific images that meet specific customer needs. This cloud controller is very similar to k8s controller. IaaS platforms now provide containers and VMs. This container orchestration is very similar to what is done in Kubernetes but the difference is that it is used for third party customer requests.

Question: In the first part of the lab 1, if the number of requests cross the length of the bounded buffer, what needs to be done?

Answer: Ideally, one could just one wait. The producer thread will wait to put the request in the queue and all the incoming requests would be blocked at the socket level. This will be handled at the OS level and need not be handled. However, requests can be dropped using a programmatic logic but this is not required for this lab.

Question: If the main server thread is blocked by a full bounded buffer queue, what happens to requests from other clients.

Answer: If the server is blocked in such a way, all the later incoming requests are blocked at the OS level. They are put in an OS buffer. This OS level buffer can fail if the load injected crosses a threshold but this would generally not happen.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 12: March 07

Lecturer: Prashant Shenoy *Scribe:* ***Ritvika Pillai***

12.1 Overview

The topic of the lecture is "Time ordering and clock synchronization." This lecture covered the following topics:

Clock Synchronization: Cristian's algorithm, Berkeley algorithm, NTP, GPS

Logical Clocks: Event ordering

12.2 Clock Synchronization

12.2.1 The motivation of clock synchronization

Just like usual clocks, systems have a clock that tells time to applications running on the system. Centralized machines have just 1 clock, but in the case of distributed systems each machine has its own clock. All these clocks might not be in sync and may drift over time. Hence the time will be dependent on which local clock is being checked.

For example, you modify files and save them on one machine A, and use another machine B to compile the files modified. If machine B has a faster clock than machine A, you may not correctly compile the files modified because the time of compiling files on machine B may be later than the time of editing files on machine A according to local timestamp on different machines, thus leading to errors.

12.2.2 How physical clocks and time work

1) One approach is to use astronomical metrics (solar day) to tell time. For example, solar noon is the time that sun is directly overhead. "Noon" on our clocks is different from solar noon. Noon depends on time zone while solar noon is a fixed physical fact. We typically use the notion of solar day to tell there are 24 hours between the time that sun is directly overhead on a particular location. Although this method was used for centuries, it is not accurate since it based on the length of a day.

2) Atomic clocks use the properties of atoms to measure time. Atomic clocks are the most accurate clocks and other clocks derive from this time. Typically, you will have some centralized atomic clock broadcast its time. The receivers, which may use less accurate mechanisms, are then synchronized with the atomic clock. For example, cell-phone clock also uses atomic clock to synchronize time with cell-phone broadcast tower. Some satellites also broadcast time based on atomic clocks.

3) Coordinated universal time (UTC) is based on noon in Greenwich (UK). All time zones are offset by UTC. Most of the atomic clocks broadcast UTC time regardless of timezone using wireless channels, satellites, FM radios, etc. Receivers listen to this and set local time based on the broadcast time.

4) Mechanical clock are less accurate—the accuracy is roughly one part per million. Computers typically use mechanical clocks. This small amount of inaccuracy results in clock drift because the property of the physical mechanism (usually quartz) can change with environmental properties such as temperature or humidity. To avoid clock drift, we need to synchronize machines with a master or with one another.

12.2.3 Drift tolerance and frequency of synchronization

Actual clocks have drift and hence need synchronization once in a while. This drift needs to be calculated to determine how frequently synchronization is needed.

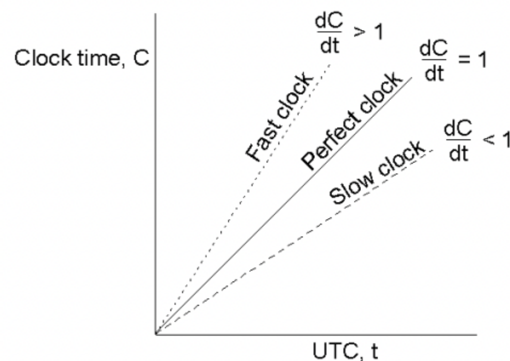


Figure 12.1: Clock drift relative to a perfect clock.

In Figure 12.1, Here, t is UTC time, C is clock time, and slope, dC/dt, is the rate of advancement of the clock. ρ indicates the inaccuracy of the clock. Consider the following cases: a. If the $dC/dt = 1$, the real time and clock advances proportionally and are in sync. b. If the $dC/dt < 1$, real time advances by 1 second and the clock will advance by $(1 - \rho)$ second, i.e., the clock runs slower. c. If the $dC/dt > 1$, real time advances by 1 second then clock will advance by $(1 + \rho)$ second, i.e., the clock runs faster. To limit the error in clock to δ, we need to synchronize every $\delta/2\rho$ seconds.

12.3 Centralized clock synchronization algorithms

12.3.1 Christian's Algorithm

In Cristian's Algorithm, is a master machine called *time server* which is the authoritative clock for telling time. It is in sync with the atomic clock via a UTC receiver. Other machines in the system synchronize with the time server.

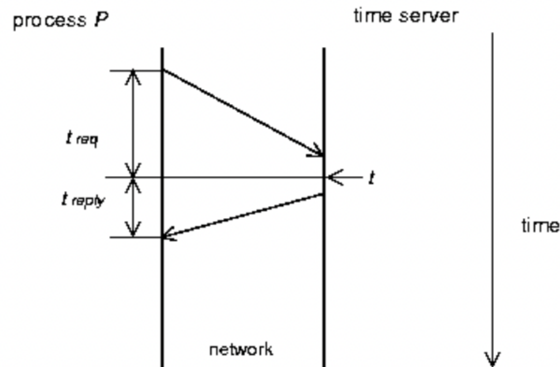


Figure 12.2: Cristian's Algorithm.

Machine P sends message to the time server to check the current time. After taking some time (t_{req}) to propagate, the request reaches the time server and will then be processed. The time server then returns the current time (t) and machine P uses this time to reset its clock. The machine P will set its time as ($t+t_{reply}$) and not just t. This is done so as to take the propagation delay from server to machine P into account. We can use $(t_{req} + t_{reply})/2$ as an estimation of t_{reply}. The better the estimate, the better the synchronization.

12.3.2 Berkeley Algorithm

This algorithm doesn't use a time server. Instead, clocks are synchronized with one another in a group, and no machine in this group synchronize with external atomic clock. We use leader election to select a "master" in a group to run clock synchronization while others are "slaves." This master clock is known as the coordinator. Each machine sends their local time to the coordinator. The coordinator then calculates an average of these times. Based on the value of average, the time of all clocks are adjusted. For example, three machines reply with their clock values as time difference of 0, -10, +25 at 3:00, then the master will tell all those machines to set their clock at $3:00 + 5(5 = (0 - 10 + 25)/3)$. This is a relative clock synchronization algorithm, not an absolute synchronization algorithm. The propagation times are estimated in the same way as Cristian's algorithm.

12.4 Distributed clock synchronization approaches

Both Cristian's and Berkley are centralized algorithms. Apart from these, there are also decentralized algorithms using resynchronized intervals. In a decentralized version of Berkley, the role of coordinator is eliminated. Instead, all machines broadcast their times to all other machines at the start of the interval. At every machine, suppose n clock values are received within the interval. Then at the end of period S, their average is calculated which is then used to set their local time. For the outliers, machines can throw away few highest and lowest values to avoid negative influence of extremely fast or slow clocks relative to the average time.

There are two decentralized approaches in use today. One approach is using NTP which is used by most computers. It uses a time server and advanced techniques to deal with network propagation delays. The accuracy is typically between 1 and 50ms. The other approach is rdate, which synchronizes a machine with

a specified machine. In many cases, you can run rdate with the argument of the name of server and just synchronize clock with that server.

12.4.1 Network Time Protocol (NTP)

NTP is widely used standard which based on Cristian's algorithm. In NTP clock synchronization, you also want to find out network propagation delay (dT_{res}). NTP clock synchronization uses a hierarchical protocol and unlike Cristian's algorithm, it does not let the clock be set backward. Since the fast clock cannot go backward, it is synchronized by slowing it down. Letting a clock go backward can have many negative consequences (such as two files having the same timestamp). This is the reason why NTP is widely used compared to Cristian's algorithm.

12.4.2 Global Positioning System (GPS)

GPS is a technology that allows any device to figure out its location. It requires clock synchronization to accurately figure out where the device is located. For example, a phone has a GPS chip that listens to satellite broadcasts. Uses the principle of triangulation to know where you are with respect to the known position of the satellite. These known positions are called landmarks. GPS achieves high accuracy because it is synchronized with satellites which use atomic clocks without a heirarchical protocol.

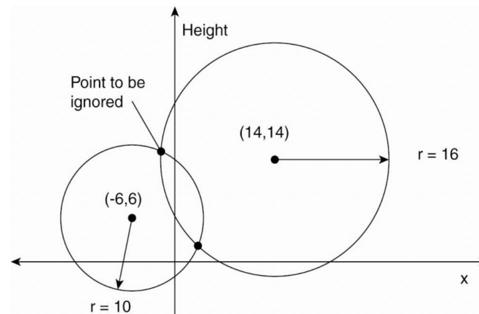


Figure 12.3: Global Positioning System (GPS)

2D space: Let the 2 landmarks be (14,14) and (-6,6). A device somewhere in this space will measure its distance with respect to these landmarks. Say, it is 16 units from the first landmark and 10 units from the second, this means that the device is on the intersection of the 2 circles because it has to satisfy both the distance constraints. If a 3rd landmark is added, then the exact position of the device can be known.

3D space: We assume GPS landmark A with its position (x_1, y_1, z_1) and its timestamp t_1, and GPS receiver B (e.g. a car) with its unknown position (x, y, z) and the timestamp t receiving broadcast t_1 message from a GPS landmark. Then the distance between A and B is $di = \sqrt{x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2}$, and di also equals $c(t_2 - t_1)$, where c is the speed of light. If we assume the receiver has a drift time dr from landmark A, then we can use Equation (1) to show that $c(t_2 + dr - t_1) = \sqrt{(x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2}$. From Equation (1), we can see that there are 4 unknowns, x, y, z and dr, thus we need minimum 4 satellites to compute the location of a GPS receiver as well as its time value. If we get 4 satellites, then we can get multiple solutions of the location of the receiver. If we have 6 or 8 satellites, we can quickly narrow the solutions. Therefore, GPS does clock synchronization as well as computing the receiver's location.

12.5 Logical clock

The above approaches use timestamps to reason the order of events. If the time difference between two events is smaller than the accuracy, then we cannot say which event happens first, thus problems may be caused. In some cases, if processes need to know the order in which the events occurred instead of the exact time, then logical clocks should be used. Hence absolute time isn't important and clock synchronization isn't needed.

12.5.1 Event Ordering

In logical clocks, there is no global clock and local clocks may run faster or slower. The ordering of events needs to be figured out in such a situation. There are some key ideas of logical clocks proposed by the scientist Lamport: we can use send/receive messages exchanged between processes/machines to order events, and if 2 processes never communicate with each other, then in such cases we don't need to find order.

The happened-before relation: We use the fundamental property that the send event occurs before the receive event. The relation is transitive, i.e, if A occurs before B and B occurs before C, then A occurs before C.

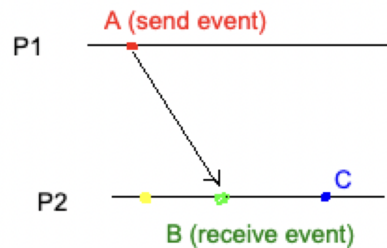


Figure 12.4: The "happened-before" relation. Note that we cannot say anything about the relation with the yellow event.

Each processor has a logical clock which gets incremented whenever an event occurs. Suppose when process i sends a message to process j, it piggybacks its local timestamp (say LCi=3) along with the message. The receiver takes this timestamp and its local timestamp (say LCj=4). Then the maximum of both these values is calculated and incremented by 1, i.e, max(LCi, LCj) + 1. This makes sure that the timestamp assigned to the receiver event is higher than the sending event. This technique was invented by Leslie Lamport.

The above algorithm only solves half the problem as it gives only forward property and not the reverse property (i,e if $timestampA < timestampB$ then A has occurred before B). Hence further changes are needed in this approach.

CMPSCI 677 Operating Systems Spring 2022

Lecture 13: March 9

Lecturer: Prashant Shenoy *Scribe:* ***Oindrila Saha***

13.1 Logical and Vector Clocks

13.1.1 Recap from last lecture

For logical clocks, the problem we address is how to reason about the ordering of events in a distributed environment without the specific times at which events occurred. We can use the concept of a logical time, which depends on message exchanges among processes, to figure out the event order. A message involves two events: the sending of a message on the first process, and the receipt of the message in the second process. Since the message has to be sent before it is received, those two events are always ordered. This allows us to order two events across machines. In addition, local events within a process are also ordered based on the execution order. According to the transitive property of events ordering, events across processes are ordered.

Lamport's logical clock is a partial ordering of events. Specifically, the events that occur after the send cannot be ordered w.r.t. another process unless there are more messages exchanged. Similarly, events that occur before the receipt of a message also cannot be ordered w.r.t. other processes unless there are other prior messages that have been received.

Another disadvantage of logical clocks is that it does not have causality. If event A has occurred before event B, then the clock value of A is less than the clock value of B. However, if a clock value is numerically less than the other, we cannot say the first event has happened before the second one. In Lamport's clock if events are concurrent we cannot assign any ordering to them as there is no happen before relationship. Therefore, we only get a partial order using logical clocks.

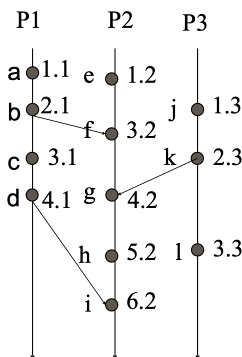


Figure 13.1: Creating total order by appending process id

13.1.2 Total Order

To convert a partial order into a total order, we can impose an arbitrary order by appending '.' and a process' id to a logical time value in each process. As a result, the process id can be used to break ties. What requires attention is that the total order is just a tie-breaking rule to assign an order for the events, so it does not actually tell us the real order of events. All the Lamport's clock properties still hold. This is an approach that many system designers used to take Lamport's clock which gives us a partial order and convert it into a set of events that gives us a total order.

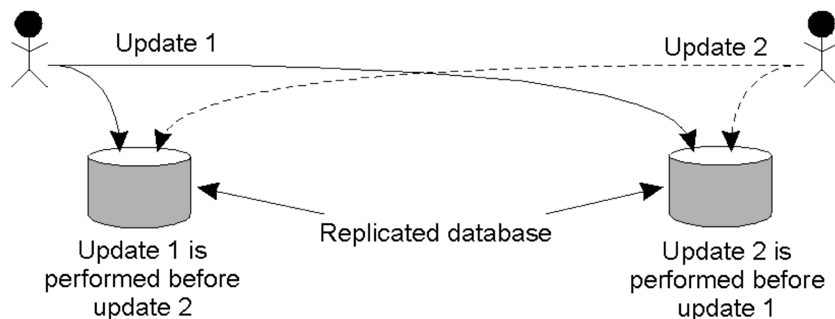


Figure 13.2: Example: Totally-Ordered Multicasting

13.1.3 Example: Totally-Ordered Multicasting

Here we use the example to show the idea of total orders. There are two replicated databases. Because they are replicated a query goes to both copies and both will update. With a partial ordering, the problem arises when there are queries that are sent in parallel. It may happen that when a user (User 1) sends a query to both replicas and is closer geographically to one of them (thus causing the query to be recieved sooner in the closer replica), and another user (User 2) sends another query at the same time and is closer to the other replica, the queries are executed out of order, thus producing inconsistent results.

To ensure the ordering is the same on each replica, we can use Lamport's clock to order all the transactions by a totally-ordered multicasting approach. When a set of transactions comes in to any replica, we use a logical value to clock them similarly with the above idea of total order.

In this example, whenever a message is sent to one database, it is also sent to all the other databases. All queries and transactions are replicated. So messages are multicast to all the database replicas as shown in Figure 13.2. Like the total order example, the logical time in each machine can be appended with "." and its machine id, allowing us to break ties for the logical clock values across different machines. Thus, each database replica will respect to a consistent order.

The details of this algorithm are not expended too much in the lecture, but the totally-ordered multicasting approach can ensure each replica receives all the transactions and executes them in a consistent order.

Q: Why is there no 2.2 in Figure 13.1
A: When you send a message, you always piggyback the logical clock value. When b is sent it takes max of 1.1 and 2.1 and you get 2.1 and add 1 to get 3.1. You can skip Ids; it does not need to be sequential.

13.1.4 Causality

In Lamport's algorithm we couldn't say that if the clock value of A is less than B, then A has happened before B. Nothing can be said about events by comparing timestamps. In some cases, we may need to maintain causality, i.e., if A happened before B, A is causally related to B. *Causal delivery* means that if the send of a message happens before another message, the receive should also be in this order. We need a time stamping mechanism such that if T(A)¡T(B), A should have causally preceded (happened before) B.

13.1.5 Vector Clocks

Causality can be captured by means of **vector clocks**. Instead of just one integer, every process maintains a vector as long as the number of processes, so each process i has a vector V_i. $V_i[i]$ is the number of event that have occured at i, which is effectively its Lamport's clock. $V_i[j]$ is the number of events i knows have occured at process j. Vector clocks are updated as follows in each of the below situations:

- Local event: increment $V_i[I]$
- Send a message: piggyback entire vector V
- Receipt of a message: $V_j[k] = \max(V_j[k], V_i[k])$
- Receiver is told about how many events the sender knows occurred at another process k
- Also $V_j[j] = V_j[j] + 1$

The idea of how vector clocks works is shown in Figure 13.3. Assume we have processes P_1, P_2 and P_3. Each process keeps an integer logical clock and the logical clock will be ticked whenever a local event happens. However, rather than keeping a single clock value like before, each process maintains a vector of clock value, one for each process in the system. Essentially, all clocks get initialized to `(0,0,0)`. The three elements are for process P_1, P_2, and P_3 respectively. Every time an event occurs in process `i` (`i` can be P_1, P_2 or P_3), the `i-th` element of process `i`'s vector gets incremented. E.g., process P_1 get its logical clocks in the first element of its vector clock incremented in the first events. When a message is sent by process `i`, the vector clock is piggybacked onto that message. When a process `j` received the message, the `i-th` element will be the maximum of `i-th` element of vector clock `i` and local vector clock `j`, and the `j-th` element will get incremented by one.

Two examples are shown in Figure 13.3 when a message is passed from P_1 to P_2 and V_2 is updated to `(1,2,0)` and then when a message is passed from P_2 to P_3 and V_3 is updated to `(1,3,4)`. Through sequence of messages each vector clock knows something about the other processes either directly or indirectly.

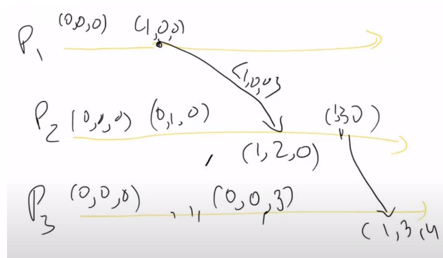


Figure 13.3: Vector Clocks

To check if there is a happen before relationship maintains lets take the send (S) from P_1 and the recieve at P_3 (R). Clearly, S has happened before R. We need to have `(1,3,4)` > `(1,0,0)`. We define greater between vectors as V1¿V2 if every element of V1 is $\geq$ every element of V2 and there is at least one element that is strictly greater. This is true in the case of S and M. We cannot compare `(1,4,0)` and `(1,3,4)`. That means they are concurrent events. Vector clocks claims the causality property.

It is worth noting that, unlike the logical clocks which are all comparable integers, vector clocks give undefined order relationship to two concurrent or independent events, which respects the reality and causality.

13.1.6 Enforcing Causal Communication

Using vector clocks, it is now possible to ensure that a message is delivered only if all messages that causally precede it have also been received as well. To enable such a scheme, we will assume that messages are multicast within a group of processes. As an example, consider three processes `P0`, `P1`, and `P2` as shown in Figure 13.4. At local time `(1,0,0)`, `P0` sends message `m` to the other two processes. After its receipt by `P1`, the latter decides to send `m*`, which arrives at `P2` sooner than `m`. At that point, the delivery of `m*` is delayed by `P2` until `m` has been received and delivered to `P2`'s application layer.

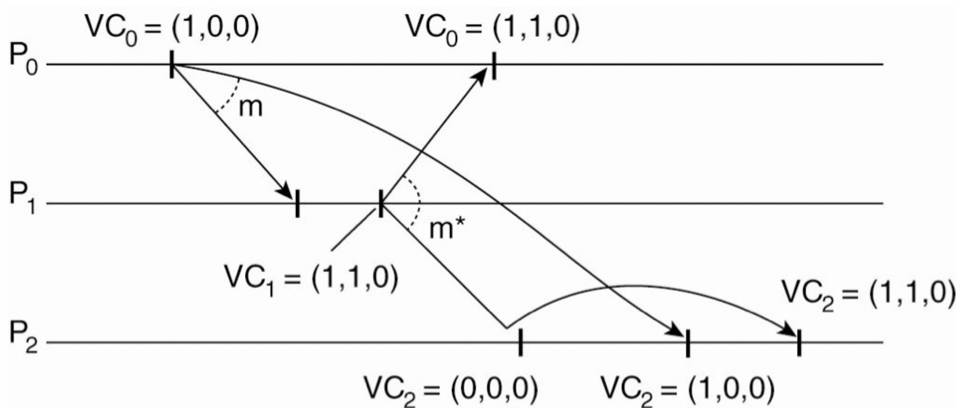


Figure 13.4: Enforcing causal communication

Q: Why is there VC1 not (1,2,0) in Figure 13.4?
A: Using the algorithm defined in this lecture it should be (1,2,0).

13.2 Global States and Distributed Snapshots

13.2.1 Global State

Problem definition: We want to run a distributed application where one of n processes crashes. Rather than killing all the processes and starting from the beginning, we can periodically take snapshots (or checkpoints) of the distributed application to keep a global state and start from the latest snapshot.

In a distributed system, when there are n processes communicating with each other, taking a checkpoint is harder. We need to take the snapshot in a consistent manner.

The global state includes the local state of each process and messages that are in transit (like the TCP buffers). The snapshot for a global state should be captured in a consistent fashion even without clock synchronization. A "consistent fashion" means that whenever restarting the computation from a checkpoint, you should get the same end result as if there was no cache at all. We will eliminate the notion of a clock and derive a technique independent of clock synchronization.

Specifically, when a message exchanging crosses the snapshot taking, the sending point of the message instead of the receipt should be captured in the state. As shown in Figure 13.5, a consistent cut (a) can achieve the consistent state while a inconsistent cut (b) can not. For (b), if you restart the computation from whenever the dotted line hits each of the processes, m2 received by P3 can be inconsistent – P3 already saw the m2 before the snapshot and it will see m2 again after re-computation from the snapshot and P2 resent it. the send should be to the left of the cut and the receive to the right.

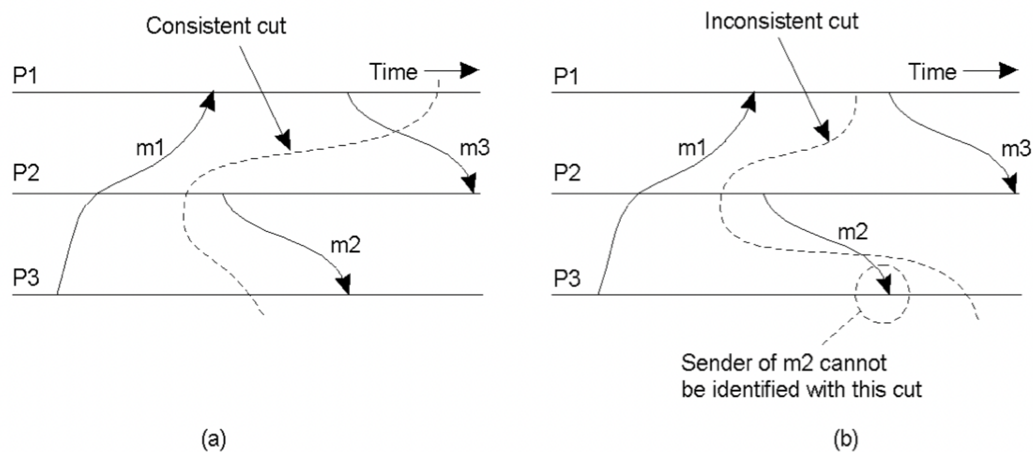


Figure 13.5: Consistent and Inconsistent cuts

13.2.2 Distributed Snapshot

A simple technique to capture is to assume a global synchronized clock and freeze all machines at the same time and capture whatever that are in all memories at the exactly same time and capture everything that is in transit. But there is no global clock synchronization. Distributed snapshotting is a mechanism that gives a consistent global state without a global clock synchronization.

The Photograph Example: Imagine a photographer taking a picture of the sky with birds flying around in it. They cannot capture entire sky in one picture, so instead they stitch pictures of the left, middle, and right parts of sky. If they take a picture of the left, and then take a picture of the middle, but a bird has flown from left to the middle, they have taken a picture of that bird twice because of inconsistency. The goal is to get a consistent photo—for it to be consistent, then all three pictures have captured all of the birds that were in the sky exactly once.

Photographs are like snapshots (in fact, this is where the name comes from). Birds are messages that are going from one process to another. You do not want a bird missing or duplicated in your picture just like you do not want a message lost or double counted.

13.2.3 Distributed Snapshot Algorithm

Assume each process communicates with another process using undirectional point-to-point channels (e.g., TCP connections). Any process can initiate the snapshot algorithm. When a process initiates the algorithm, it will first checkpoint its local state, and then send a marker on every outgoing channel. When a process receives a marker, it will have different actions depending on whether it has already seen the marker for this snapshot. If the process sees a marker at the first time, it will checkpoint its state, send markers out and start saving messages on all the other channels. If the process sees a marker for the snapshot at the second time, it will stop saving messages for the channel from which the marker comes. A process will finish a snapshot until it receives a marker on each incoming channel and processes them all.

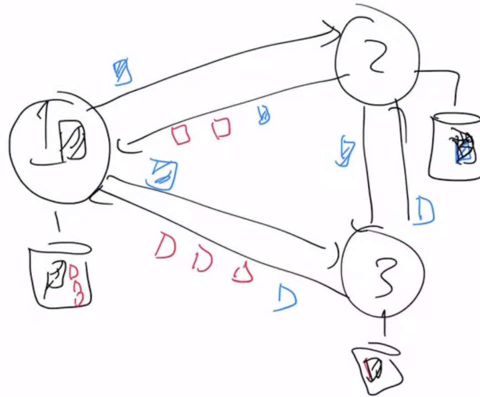


Figure 13.6: Distributed Snapshot Example

For example, in Figure 13.6, process 1, 2 and 3 communicates with each other through the duplex TCP connections. When process 1 initiate a snapshot, it will first save its memory contents (state) into the disk, and then send a marker (in blue) to 2 and 3. It will also start to save the incoming messages(in red) (TPC buffers) from 2 and 3. When 2 receives a marker from 1, it will start to checkpoint state, send a marker out to 1 and 3, and start saving messages from 3. Assume 3 sees the marker sent from 1 first, it will checkpoint its state, sends out a marker to 1 and 2, and start saving messages from 2. 1 will stop saving messages for 2 or 3 until a marker from 2 or 3 arrives. 2 will stop saving messages for 3 until a marker from 2 arrives. And P3 will stop saving messages for 2 until a marker from 3 arrives. Each process will finish this snapshot when it sees a marker from every incoming channel. Thus, a distributed snapshot captured.

13.2.4 Snapshot Algorithm Example

Q decides to take a snapshot. Consider Figure 13.7. b) It receives a marker for the first time and records its local state; c) Q records all incoming message; d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel. First incoming marker says start recording and all other incoming messages say stop recording.

Q: Is it guaranteed to be correct?
A: It is guaranteed to be correct under the assumption that all channels are TCP/IP sockets and we are delivering messages in order. It is guaranteed to terminate as well.
Q: In a distributed snapshot can you have a case where a send is before the cut and the receive is after?
A: In Fig this the case where you will record m3. It will keep recording incoming channels and will save m3 as part of its disk. P2 will process m3 and snapshot will save a copy on disk too.

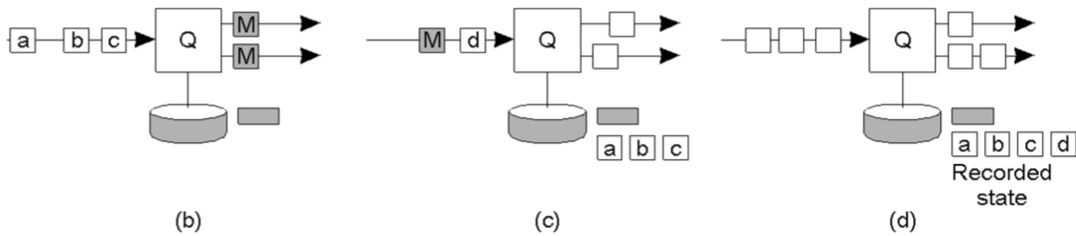


Figure 13.7: Snapshot algorithm example.

Q: Can the process not send new messages before you receive?
A: There is no restriction on what the process can do when a snapshot is happening.

13.3 Termination Detection

This involves detecting the end of a distributed computation. We need to detect this since a process cannot just exit after completing its events if another process wants to send it a message. Let sender be the predecessor, and receiver be the successor. There are two types of markers: Done and Continue. After finishing its part of the snapshot, process Q sends a Done or a Continue to its predecessor. Q can send a Done only when:

- All of Q's successors send a Done
- Q has not received any message since it check-pointed its local state and received a marker on all incoming channels
- Else send a Continue

Computation has terminated if the initiator receives Done messages from everyone.

Q: What happens when there is a failure?
A: Terminal detection algorithm does not handle failures. We need to go previous snapshot if there is a crash.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 14: March 21

Lecturer: Prashant Shenoy *Scribe:* ***Y. Vayunandhan Reddy***

14.1 Overview

This section covers the following topics:

Leader Election: Bully Algorithm, Ring Algorithm, Elections in Wireless Networks

Distributed Synchronization: Centralized, Decentralized, Distributed algorithms

Chubby Lock Service

14.2 Leader Election

Many tasks in distributed systems require one of the processes to act as the *coordinator*. Election algorithms are techniques for a distributed system of N processes to elect a coordinator (leader). An example of this is the Berkeley algorithm for clock synchronization, in which the coordinator has to initiate the synchronization and tell the processes their offsets. A coordinator can be chosen amongst all processes through leader election.

14.2.1 Bully Algorithm

The bully algorithm is a simple algorithm, in which we enumerate all the processes running in the system and pick the one with the highest ID as the coordinator. In this algorithm, each process has a unique ID and every process knows the corresponding ID and IP address of every other process. A process initiates an election if it just recovered from failure or if the coordinator failed. Any process in the system can initiate this algorithm for leader election. Thus, we can have concurrent ongoing elections. There are three types of messages for this algorithm: *election*, *OK* and *I won*. The algorithm is as follows:

1. A process with ID i initiates the election.
2. It sends *election* messages to all process with ID > i.
3. Any process upon receiving the election message returns an OK to its predecessor and starts an election of its own by sending *election* to higher ID processes.
4. If it receives no OK messages, it knows it is the highest ID process in the system. It thus sends *I won* messages to all other processes.
5. If it received OK messages, it knows it is no longer in contention and simply drops out and waits for an *I won* message from some other process.
6. Any process that receives *I won* message treats the sender of that message as coordinator.

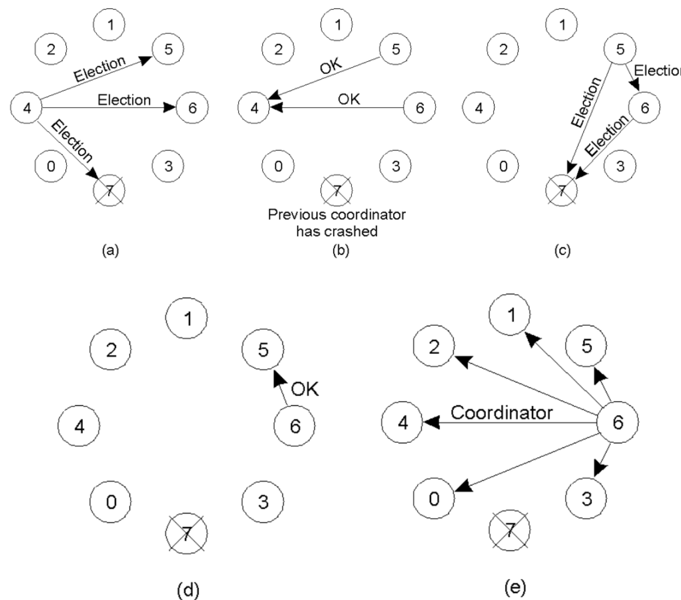


Figure 14.1: Depiction of Bully Algorithm

An example of Bully algorithm is given in Figure 14.1. Communication is assumed to be reliable during leader election. If the communication is unreliable, it may happen that the elected coordinator goes down after it being elected, or a higher ID node comes up after the election process. In the former case, any node might start an election process after gauging that the coordinator isn't responding. In the latter case, the higher ID process asks its neighbors who is the coordinator. It can then either accept the current coordinator as its own coordinator and continue, or it can start a new election (in which case it will probably be elected as the new coordinator). This algorithm runs in $O(n^2)$ time in the worst case when lowest ID process initiates the election. The name bully is given to the algorithm because the higher ID processes are bullying the lower ID processes to drop out of the election.

Question: What happens if 7 has not crashed? Who will it send message to?
Answer: Suppose if 7 has not crashed in the example, it would have sent the response when 4 has started the election. 4 would have dropped out and the recursion would have continued and 7 would have elected as leader finally.

Question: Can 7 never initiate the election?
Answer: If 7 is already a leader there is no reason for it to initiate an election.

Question: When does a smaller ID process know it should start an election?
Answer: This is not particularly specified by this algorithm. Ideally this is done when it has not heard from the coordinator in a while (timeout period).

Question: In the above example what happens if 7 is recovered?
Answer: Any process that is recovered will initiate an election. It will see 6 is the coordinator. In this case 7 will initiate an election and will win.

Question: In the example, how will 7 see 6 is the coordinator (How does a process know who the coordinator is)?
Answer: Discovering who is the coordinator is not part of the algorithm. This should be implemented separately (storing it somewhere, broadcasting the message).

Question: What happens when we have a highly dynamic system where processes regularly leave and join (P2P system)?
Answer: The bully algorithm is not adequate for all kinds of scenarios. If you have a dynamic system, you might want to take into account the more stable processes (or other metrics) and give them higher ids to have them win elections.

14.2.2 Ring Algorithm

The ring algorithm is similar to the bully algorithm in the sense that we assume the processes are already ranked through some metric from 1 to n. However, here a process i only needs to know the IP addresses of its two neighbors (i+1 and i-1). We want to select the node with the highest id. The algorithm works as follows:

- Any node can start circulating the election message. Say process i does so. We can choose to go clockwise or counter-clockwise on the ring. Say we choose clockwise where i+1 occurs after i.
- Process i then sends an election message to process i+1.
- Anytime a process j $\neq i$ receives an election message, it piggybacks its own ID (thus declaring that it is not down) before calling the election message on its successor (j+1).
- Once the message circulates through the ring and comes back to the initiator i, process i knows the list of all nodes that are alive. It simply scans the list and chooses the highest id.
- It lets all other nodes about the new coordinator.

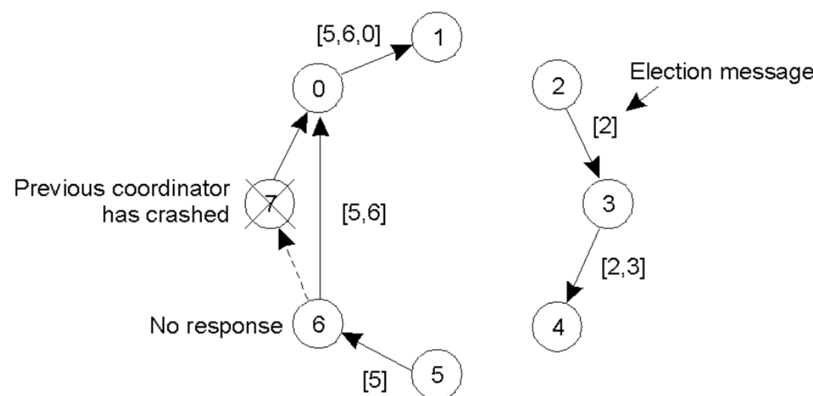


Figure 14.2: Depiction of Ring Algorithm

An example of Ring algorithm is given in Figure 14.2. If the neighbor of a process is down, it sequentially polls each successor (neighbor of neighbor) until it finds a live node. For example, in the figure, when 7 is down, 6 passes the election message to 0. Another thing to note is that this requires us to enforce a logical ring topology on the underlying application, i.e. we need to construct a ring topology on top of the whole system just for leader election.

Question: Does every process need to know about the network topology?
Answer: Every process know the IDs and IP addresses of other processes (assumption). There is no real topology, from the table we can find the neighbours.

Question: If we already know the IDs of all processes why there is a need to go around the ring?
Answer: There can be different kind of failures, e.g., the process may crash or the network may crash while partitioning the ring. We do no know how many processes have disconnected from the ring. We need to actually query and check what is the issue.

Question: How does 6 know that it has to send message to 0 when 7 is down?
Answer: Here we can assume that every node not only has information of its neighbors, but neighbors' neighbors as well. In general, in a system where we expect a max of k nodes to go down during the time leader election takes to execute, each node needs to know at least k successive neighbors in either direction. This is still less than what each node needs to know in Bully algorithm.

14.2.3 Time Complexity

The bully algorithm runs in

- $O(n^2)$ in the worst case (this occurs when the node with lowest ID initiates the election)
- $O(n-2)$ in the best case (this occurs when the node with highest ID that is alive initiates the election)

The ring algorithm always takes 2(n-1) messages to execute. The first (n-1) is during the election query and second time to announce the results of the election. It is easy to extend ring algorithm for other metrics like load, etc.

Question: How do you know a node is not responding?
Answer: If it has actually crashed then TCP will fail while setting up socket connection. Otherwise, it can be a slow machine which is taking time. It is a classical problem in distributed systems to distinguish between a slow process and a failed process which is a non-trivial problem. Timeout is not an ideal solution but can be used in practice.

14.3 Distributed Synchronization

Every time we wish to access a shared data structure or critical section in a distributed system, we need to guard it with a lock. A lock is acquired before the data structure is accessed, and once the transaction has completed, the lock is released. Consider the example below:

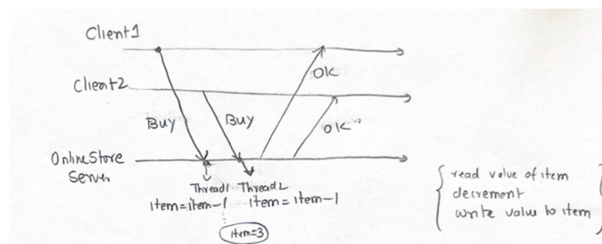


Figure 14.3: Example of a race condition in an online store.

In this example, there are two clients sending a buy request to the Online Store Server. The store implements a thread-pool model. Initially the item count is 3. The correct item count should be 1 after two buy operations. If locks are not implemented there may be chance of race condition and item count can be 2.

This is because the decrement is not an atomic operation. Each thread needs to read, update and write the item value. The second thread might read the value while first thread is updating the value (it will read 3) ans update it to 2 and save it, which is incorrect. This is an example of trivial race condition.

14.3.1 Centralized Mutual Exclusion

In this case, locking and unlocking coordination are done by a master process. All processes are numbered 1 to n. We run leader election to pick the coordinator. Now, if any process in the system wants to acquire a lock, it has to first send a lock acquire request to the coordinator. Once it sends this request, it blocks execution and awaits reply until it acquires the lock. The coordinator maintains a queue for each data structure of lock requests. Upon receiving such a request, if the queue is empty, it grants the lock and sends the message, otherwise it adds the request to the queue. The requester process upon receiving the lock executes the transaction, and then sends a release message to the coordinator. The coordinator upon receipt of such a message removes the next request from the corresponding queue and grants that process the lock. This algorithm is fair and simple.

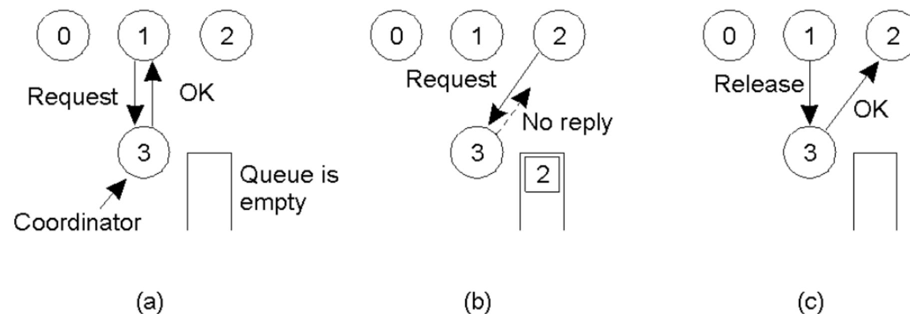


Figure 14.4: Depiction of centralized mutual exclusion algorithm.

An example of the algorithm is given in Figure 14.4. There are two major issues with this algorithm, related to failures. When coordinator process goes down while one of the processes is waiting on a response to a lock request, it leads to inconsistency. The new coordinator that is elected (or reboots) might not know that the earlier process is still waiting for a response. This issue can be tackled by maintaining persistent data on disk whenever a queue of the coordinator is altered. Even if the process crashes, we can read the file and persist the state of the locks on storage and recover the process.

The harder problem occurs when one of the client process crashes while it is holding the lock (during one of its transactions). In such a case, coordinator is just waiting for the lock to be released while the other process has gone down. We cannot use timeout in this case, because usually transactions take arbitrary amount of time to go through. All other processes that are waiting on that lock are also blocked forever. Even if the coordinator somehow knew that the client process has crashed, it may not always be advisable to take the lock forcibly back because the client process may eventually reboot and think it has the lock and continue its transaction. This causes inconsistency. This is a thorny problem which does not have any neat solution. This limits the practicality of such an centralized algorithm.

14.3.2 Decentralized Algorithm

Decentralized algorithms use voting to figure out which lock requests to grant. In this scenario, each process has an extra thread called the coordinator thread which deals with all the incoming locking requests.

Essentially, every process keeps a track of who has the lock, and for a new process to acquire a new lock, it has to be granted an *OK* or go ahead vote from the strict majority of the processes. Here, majority means more than half the total number of nodes (live or not) in the system. Thus, if any process wishes to acquire a lock, it requests it from all other processes and if the majority of them tell it to acquire the lock, it goes ahead and does so. The majority guarantees that a lock is not granted twice. Upon the receipt of the vote, the other processes are also told that a lock has been acquired and thus, the processes hold up any other lock request. Once a process is done with the transaction, it broadcasts to every other process that it has released the lock.

This solves the problem of coordinator failure because if some nodes go down, we can deal with it so long as the majority agrees that whether the lock is in use or not. Client crashes are still a problem here.

14.3.3 Distributed Algorithm

This algorithm, developed by Ricart and Agrawala, needs $2(n-1)$ messages and is based on Lamport's clock and total ordering of events to decide on granting locks. After the clocks are synchronized, the process that asked for the lock first gets it. The initiator sends request messages to all $n-1$ processes stamped with its ID and the timestamp of its request. It then waits for replies from *all* other processes.

Any other process upon receiving such a request either sends reply if it does not want the lock for itself, or is already in the transaction phase (in which case it doesn't send any reply and the initiator has to wait), or it itself wants to acquire the same lock in which case it compares its own request timestamp with that of the incoming request. The one with the lower timestamp gets the lock first.

- Process k enters critical section as follows:
  - Generate new time stamp $TS_k = TS_{k+1}$
  - Send request(k,TS_k) all other n-1 processes
  - Wait until reply(j) received from all other processes
  - Enter critical section
- Upon receiving a request message, process j
  - Sends reply if no contention
  - If already in critical section, does not reply, queue request
  - If wants to enter, compare TS_j with TS_k and send reply if $TS_k < TS_j$, else queue (recall: total ordering based on multicast)

This approach is fully decentralized but there are n points of **failure**, which is worse than the centralized one.

14.3.4 Token Ring Algorithm

In the *token ring algorithm*, the actual topology is not a ring, but for locking purpose there is a logical ring and processes only talk to neighboring processes. A process with the token has the lock at that time. To acquire the lock one needs to wait. The token is circulated through the logical ring. No method is there to request the token, the process needs to wait to get a lock. Once the process has token it can enter the critical section.

This was designed as part of a networking protocol Token Ring. In physical networking, only one node can transmit at a time. If multiple nodes transmit at a time there is a chance of collision. Ethernet handled it by detecting collisions. A node transmits and if there is a collision it backs off and will succeed eventually. In Token Ring this was handled using locks. Only one machine has lock at an instance in the network and it transmit at that particular time.

In this algorithm one problem is loss of token. Regenerating the token is non-trivial, as you can not use timeout strategy.

Question: In a Token Ring, when should you hold the token?
Answer: If a process want to send data on the network it will wait until it receives the token. It will hold the token until it completes the transmission and pass the token to the next. If it do not require the token it simply passes it.

Question: Is the token generated by the process?
Answer: A token is special message that is circulating in the network. It is generated when the system started. A process can hold the message or pass it to the next.

Question: In Token Ring, if the time limit is finished and the process is still in the critical section, what happens?
Answer: In general network transmission this can limit the amount of data you can transmit. But if this is used for locking the message can be held for arbitary amount of time (based on the critical section). This may complicate the token recovery because we can't distinguish if a token is lost or any process is in long critical section.

14.4 Chubby Lock Service

This was developed by Google to provide a service that can manage lots of locks for different subgroups of applications. Each Chubby cell has group of 5 machines supporting 10,000 servers (managing the locks of applications running on them). This was designed for coarse-grain locking with high reliability. One of the 5 machines is maintained outside the data center for recovery.

Chubby uses distributed lock to elect a leader. The process with the lock is elected as leader. The 5 processes in the cell run an internal leader election to elect a primary. The other are lock workers. These are used for replication. The applications in the system ask for locks and release locks using RPC calls. All of the locks requests go to the primary. The lock state is kept persisted on disk by all the machines. This uses a file abstraction for locks. To lock and unlock, file is locked and unlocked respectively. State information can also be kept in the file. It supports reader-writer locks. If the primary fails it triggers a new leader election.

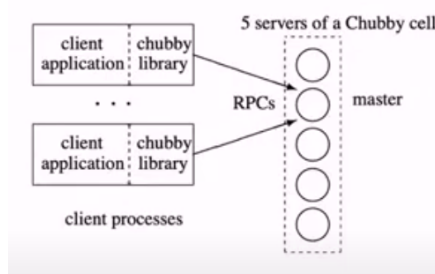


Figure 14.5: Chubby lock service.

Question: What is the purpose of multiple replicas?
Answer: To ensure fault tolerance and lock state doesn't disappear.

Question: Do the replicas have any purpose till the primary fails?
Answer: They store all the lock information in the database. They work as hot standby and can takeover with current state if the primary fails.

Question: Where are the lock files created?
Answer: We have distributed file system which looks same on all the machines.

Question: If Chubby is used for leader election how does it handle coordinator failure?
Answer: It has a notion of lease, every process can hold lease for particular lease period and needs to be renewed. This ensures that the lock is never with a failed process.

Question: While writing is it through all nodes or just through master?
Answer: All client requests are directed to master. The master sends the operations performed to others to keep them in sync.

CMPSCI 677 Operating Systems	**Spring 2022**

Lecture 15: March 23

Lecturer: Prashant Shenoy *Scribe:* ***Dinesh Kandula***

15.1 Overview

This lecture covers the following topics:

Distributed Transactions: ACID properties, Transaction Primitives, Private Workspace, Write-ahead logs.

Currency control and locks: Serializability, Optimistic Concurrency Control, Two-phase Locking (2PL), Timestamp-based Concurrency Control.

15.2 Transactions

Let us try and understand Transactions and their importance using an example. Let us assume there are two clients, client 1 and client 2, which are trying to make a transaction on bank accounts A, B, C. Client 1 wants to transfer $4 from account A to account B and client 2 wants to transfer $3 from account C to account B. In the end, $7 needs to be deposited into account B. Let's assume accounts A, B, C has $100, $200, $300 respectively initially. To transfer, client 1 needs to read and deduct balance in account A and then transfer by reading and updating balance in account B. Similarly, client 2 needs to read and deduct balance in account C and then transfer by reading and updating balance in account B.

Let us say if client 1 and client 2 makes RPC calls to bank's database to perform their respective operations at the same time. There are many possible ways all of the operations from client 1 and client 2 could be interleaved. Figure 15.1 shows one possible interleaving. As shown in Figure 15.1, initially client 1 reads, deducts and updates the balance in A. Next, client 2 reads, deducts and updates the balance in C. In the next step, client 1 reads balance in account B and then client 2 reads and add $3 to the balance in account B. But client 1 still has the old value and it adds $4 to old value and updates the balance in account B by overwriting the changes made by client 2. In the end, only $4 were transferred to account B instead of $7. This interleaving gave us incorrect result.

An important thing to note is that this database is shared by multiple clients. To solve the above issue, you could try fine-grained locking by locking the account before performing a read or write and then release the lock once the operation is done. This is still not going to solve the issue as the same interleaving can still happen. Instead, by locking the entire database and performing all of the operations each client needs to do before releasing the lock, this issue can be solved. But locking entire database will prevent other clients from performing any operations while one client is executing. This will degrade the performance significantly.

Question: If you lock the entire object and perform all of the operations on that object before releasing it, could this solve the above issue instead of locking the entire database?
Answer: Locking an object across multiple operations is not fine-grained locking. Fine-grained locking, which is locking and releasing for each operation, cannot the solve the above issue. Coarser-grained locking

Client 1	Client 2
Read A: $100	
Write A: $96	
	Read C: $300
	Write C:$297
Read B: $200	
	Read B: $200
	Write B:$203
Write B:$204	

Figure 15.1: Depiction of sequence of transactions by two clients

which involves locking the database or row across multiple operations can solve the issue but can degrade the performance.

Transactions are the higher level mechanism which provides atomicity of processing. *Atomicity* is when a set of operations is protected with the all or nothing property, i.e., either all of the operations succeed or none of them succeed. Anything that is protected by a transaction operates as one atomic operation even though there may be multiple statements.

Client 1	Client 2
Read A: $100	
Write A: $96	
Read B: $200	
Write B:$204	
	Read C: $300
	Write C:$297
	Read B: $204
	Write B:$207

Figure 15.2: Atomic transactions.

Figure 15.2 shows how you want the operations to happen. All of the operations by a particular client happen like one atomic operation. The order of which client executes first does not matter.

15.2.1 ACID Properties

- *Atomic:* All or nothing. Either all operations succeed or nothing succeeds.

- *Consistent: Consistency* is when each transaction takes system from one consistent state to another. A *consistent state* is a state where everything is correct. After a transaction, the system is still consistent.
- *Isolated:* A transaction's changes are not immediately visible to others but once they are visible, they become visible to the whole world. This is also called the *serializable* property. This property says that even if multiple transactions are interleaved, the end result should be same as if one transaction occurred after another in a serial manner.
- *Durable:* Once a transaction succeeds or commits, the changes are permanent, but until the transaction commits, all of the changes made can be reverted.

15.2.2 Transaction Primitives

Special primitives are required for programming using transactions. Primitives are supplied by the operating system or by the language runtime system.

- BEGIN_TRANSACTION : Marks the start of transaction.
- END_TRANSACTION : Terminate the transaction and try to commit. Everything between begin and end primitive will be executed as one atomic set of instructions.
- ABORT_TRANSACTION : Kill the transaction and undo all of the changes made.
- READ : Read data from a file, a table, or otherwise.
- WRITE : Write data to a file, a table, or otherwise.

15.2.3 Distributed Transactions

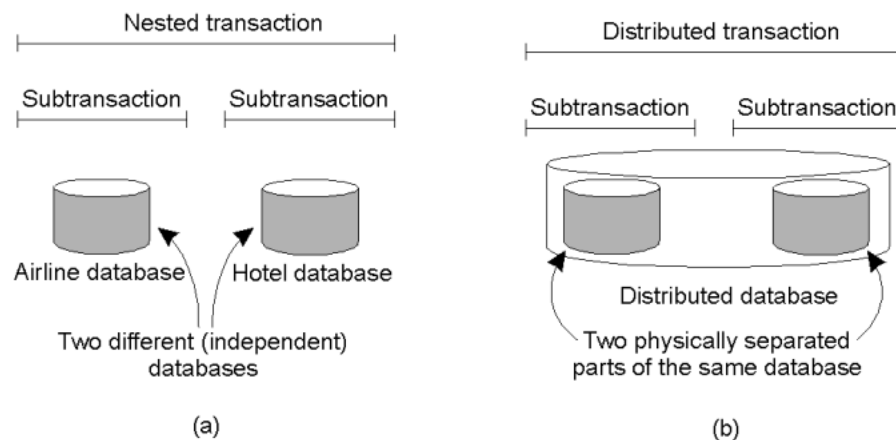


Figure 15.3: Distributed transactions.

a) *Nested Transaction*

Take an example of making a reservation for a trip which includes airline and hotel reservations. Assume you want to either do both flight and hotel booking or neither. Usually airlines and hotels are different

companies and have their own databases. This can be achieved using *nested transactions*. There are smaller transactions inside the main BEGIN_TRANSACTION and END_TRANSACTION primitives. This way, if one booking fails, you undo the changes made for other booking. So, the smaller transactions protects each booking and the bigger transaction gives ACID properties as a whole. If any one small transaction fails, the complete transaction is aborted.

b) *Distributed Transaction*

A transaction is distributed if the operations are being performed on data that is spread across two or more databases. From user's perspective, there is only one logical database. So, to make a transaction on this logical database, we will have subtransactions. Each subtransaction perform operations on a different machine. Performing operations on distributed database needs distributed lock which makes implementation difficult.

Question: Is it possible that one transaction is successful in one database and fails in a mirror database?
Answer: A distributed database is one in which data are partitioned into multiple databases instead of one database being a mirror (exact copy) of the other.

15.2.4 Implementation

Private Workspace

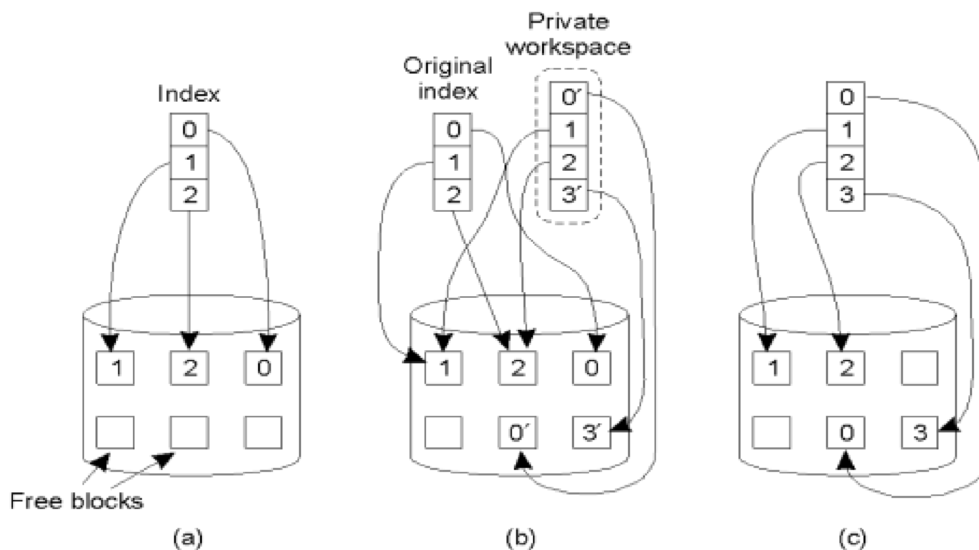


Figure 15.4: Distributed transactions.

- Every transaction gets its own copy of the database to prevent one transaction from overwriting another transaction's changes. Instead of a real copy, each transaction is given a snapshot of the database, which is more efficient. Each transaction makes changes only to its copy and when it commits, all of the updates are applied to database.
- Making a copy is optimized by using copy-on-write. A copy is not made for read operations.
- Using a copy also makes aborting a transaction easy. If a transaction is aborted, the copy is simply

deleted and no changes are applied to the original database. If a transaction is committed, you just take the changes and apply them to the database.

In Figure 15.4, the index is used to store the locations of the file blocks. To execute a transaction, instead of making copies of the file blocks, a copy of the index is made. The index initially points to original file blocks. For a transaction, it looks like it has its own copy. When the transaction needs to make an update to block 0, instead of making change to original block, a copy of the block ($0'$) is created and the change is made to the copy. Now the transaction index is made to point to the copy instead of original. In case the transaction adds something to the database, a new free block $3'$ is created. Essentially, we are optimizing by making a copy only when a write operation is executed. If the transaction is aborted, the copies are deleted. If the transaction is committed, the changes made are applied to the original database. This is the *private workspace model*.

Question: If two concurrent transactions make changes to their own copy of the same block and if the first transaction commits, does the second transaction overwrite the changes made by the first transaction?
Answer: If a transaction wants to commit and meanwhile some changes were made to the same block the transaction wants to commit to, this is a write-write conflict. In this case, the transaction is aborted.

Write-ahead Logs

In this design, the transaction make changes to the actual database. If the transaction aborts, all of the changes needs to be aborted. The value before the change is stored using an undo log. So, when the transaction aborts, the undo log is used to rollback and restore the original value. Commits need to be atomic to avoid write-write conflicts.

	Log	Log	Log
x = 0;			
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

- a) A transaction
- b) – d) The log before each statement is executed

Figure 15.5: Distributed transactions

Figure 15.5 shows a trivial transaction. The database has only two entries, x and y, initialized to 0. The transaction has three operations as shown in the figure. After the first operation, the original value of x and the update value of x is added to the log. After the second operation, another entry storing the original and new value of y is added. In the last step, again a new log storing the previous and updated value of x is added to the log. If the transaction commits, there is nothing to do as the changes were made to original

database. A commit success log is added to undo log in the end. To abort, the log is traversed backwards reverting each operation by replacing current value with the previous value.

Question: What does "force logs on commit" mean?
Answer: If transaction is committed, a entry is added to the log to indicate that the transaction is successful and there is no need for undo.

Question: What happens if multiple transactions are operating on x,y in the above example?
Answer: One approach is to use locks. While one transaction is operating, it holds a lock which prevents any other transaction from making any changes. Another approach is *optimistic concurrency control.* This approach does not use locks and assumes that transaction conflicts are rare. Conflicts are tracked and all of the transactions that are part of the conflict are aborted and restarted again.

15.3 Concurrency Control

To handle concurrency, a trivial solution is to use a lock over the complete database and allow only one transaction at a time. This solution will lead to very bad performance as only one transaction is executed at a time. So, several transactions should be allowed to be execute simultaneously for better performance. The final result after concurrent transactions should be same as if each transaction ran sequentially.

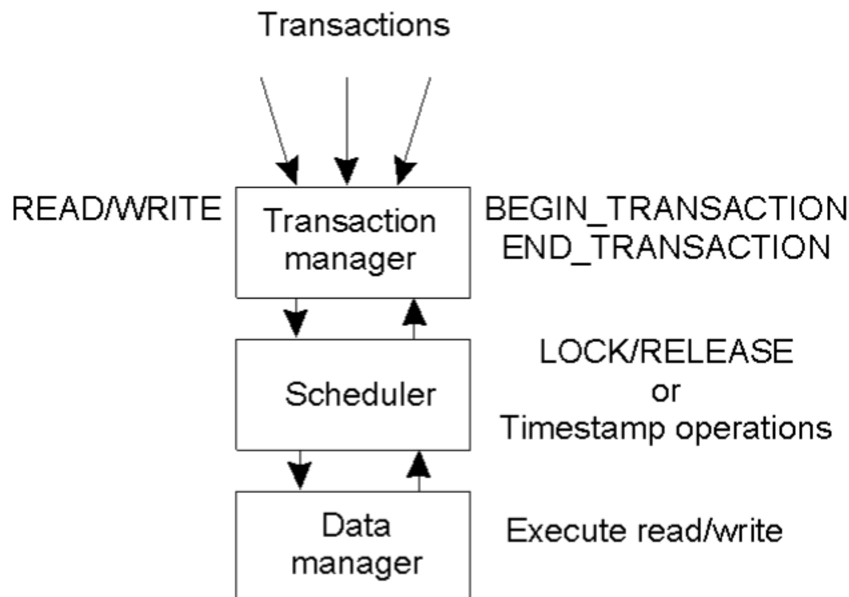


Figure 15.6: General organization of managers for handling transactions.

Concurrency can be implemented in a layered fashion as shown in Figure 15.6. The transaction manager implements the private-workspace model or write-ahead log model. The scheduler implements locking and releasing the data. The data manager makes changes to either the workspace (index) or to the actual database. In the case of distributed systems, a similar organization of managers is applied as shown in Figure 15.7. Data is now split across multiple machines. The scheduler needs to handle distributed locking now. Beyond that everything is same.

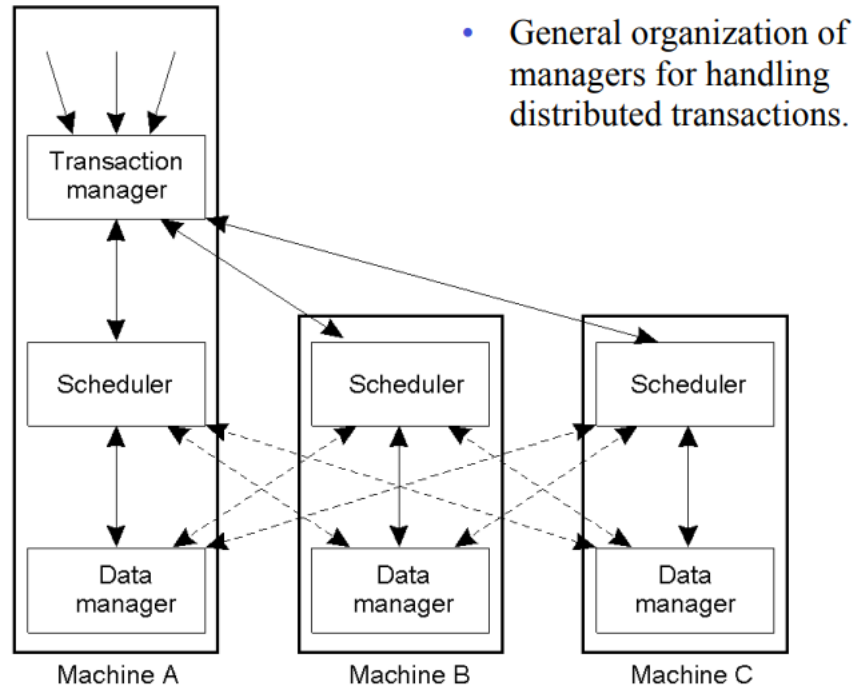


Figure 15.7: General organization of managers for handling distributed transactions.

15.3.1 Serializability

This the key property that is imposed on the end result of a transaction. The end result of a concurrent transactions should be same as if the transactions are executed serially. Figure 15.8 shows an example where each transaction modifies x, and three possible ways the transactions are interleaved. The result is valid only if it is same as the result of one possible serial orders (1,2,3 or 3,2,1 or 2,3,1 etc). If a sequential order can be found that gives the same result as the concurrent transactions, then that interleavability is considered valid.

In the example shown in Figure 15.8, Schedule 1 is valid because the output is same as if the transactions are executed in the a,b,c serial order. Schedule 2 is also valid because result is same as the result of a,b,c serial order. Schedule 3 is illegal because there is no possible sequential execution of a,b,c that gives the same result as Schedule 3. Interleaving could result in two kinds of conflicts: read-write conflicts and write-write conflicts. Read-write conflicts occurs when the transaction reads an outdated value of a variable and update it based on the outdated value. Write-write conflicts occur if one write operation overwrites another write operation's update. The scheduler should acquire the appropriate locks to prevent both of the conflicts from happening.

15.3.2 Optimistic Concurrency Control

In *optimistic concurrency control*, the transaction is executed normally without imposing serializability restriction, but the transaction is validated at the end by checking for read-write and write-write conflicts. If any conflict is found, all of the transactions involving in the conflict are aborted. This design takes an optimistic view and assumes database is large and most of the transactions occur on different parts of the

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```
(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```
(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```
(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

Figure 15.8: Example of Serializable and Non-Serializable transactions.

database. Using locks adds unnecessary overhead and this design avoids this by checking for validity in the end. It works well with private workspaces because the copies can be deleted easily if a transaction aborts.

Advantages:

- One advantage is that this method is deadlock free since no locks are used.
- Since no locks are used, this method also gives maximum parallelism.

Disadvantages:

- The transaction needs to be re-executed if it is aborted.
- The probability of conflict rises substantially at high loads because there are many transactions operating at the same time and probability of them operating on same data block is high. Throughput will go down when the load is high.

Question: If two transactions that are conflicted are aborted and re-executed again, will it not result in conflict again?
Answer: If they are executed at the same time again, they will conflict. The scheduler needs to randomize execution time or something else so that they are executed at different times and conflict is avoided.

15.3.3 Two-phase Locking (2PL)

Because the probability of conflicts is high at high loads, pptimistic concurrency control is not widely used in commercial systems. Two-phase Locking (2PL) is a standard approach used in databases and distributed systems. The scheduler grabs locks on all of the data items the transaction touches and is released at the end when the transaction ends. If a transaction touches an item and lock is grabbed, no other transaction can touch that data item. The transaction needs to wait for lock to be released if it wants to operate on locked data.

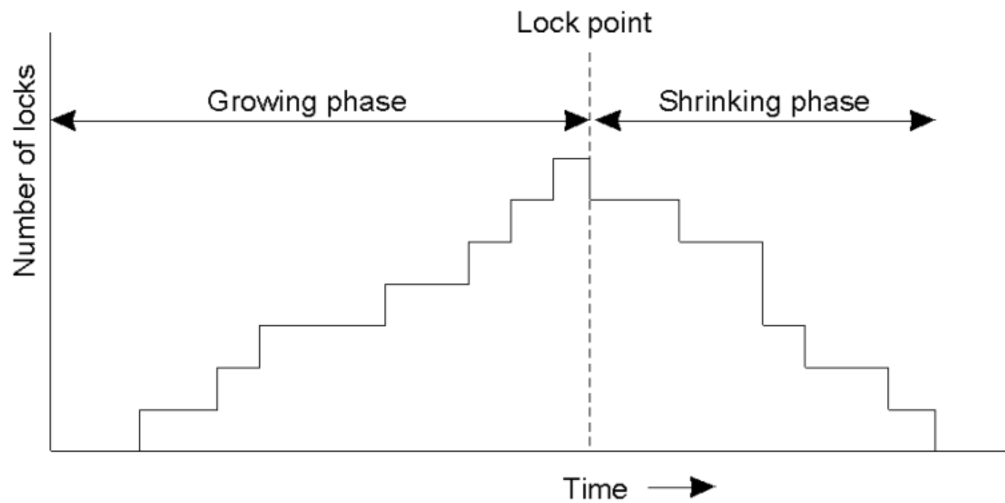


Figure 15.9: Two-Phase Locking

Additionally, a constraint is imposed that if a transaction starts releasing locks, it cannot acquire it again. This leads to two phases in each transaction as shown in Figure 15.9. During the growing phase, the transaction acquires locks and once it releases a lock, the transaction cannot acquire any more locks. This is shrinking phase as the number of locks the transaction is holding reduces. The transaction needs to make sure that it will not touch any new data before releasing first lock as it cannot acquire a lock again.

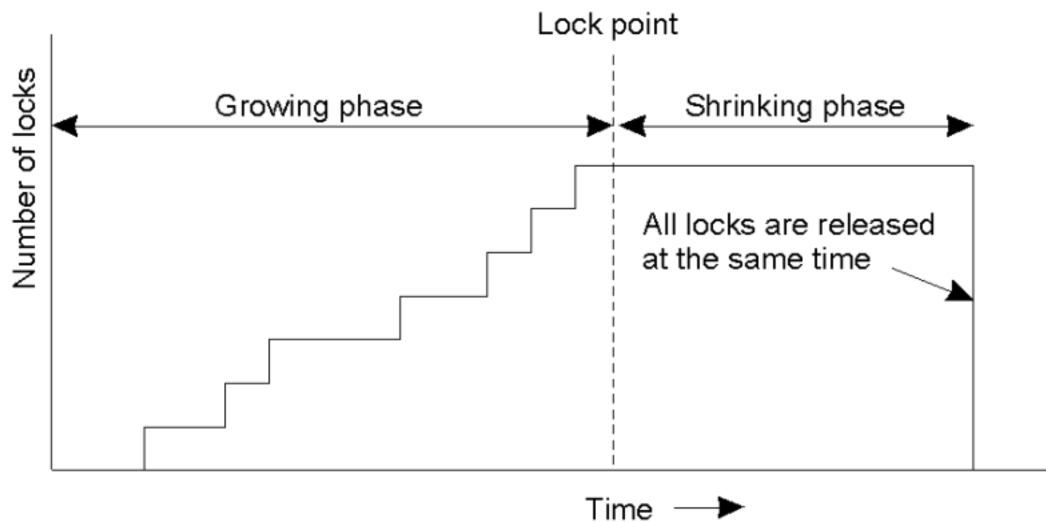


Figure 15.10: Strict Two-Phase Locking

A simpler approach is to completely avoid shrinking phase. As shown in Figure 15.10, Strict Two-Phase Locking grabs locks and releases all of the locks at a time before committing. In this method, the transaction will hold the lock until the end and does not release immediately as soon as it is done with the lock.

Question: Where is this locking mechanism implemented? Since the lock acquisition and release happens in the transaction, shouldn't it be the responsibility of the transaction manager?

Answer: The locking mechanism is implemented in the scheduler. Essentially, the transaction manager itself notifies scheduler to grab the lock on a specific data block. The transaction manager works with the scheduler to grab the locks.

15.3.4 Timestamp-based Concurrency Control

This method handles concurrency using timestamps, in particular Lamport's clock (logical clock instead of physica clock). The timestamp is used to decide the order and which transaction to abort in case of read-write or write-write conflicts. If two transactions are conflicted, the later transaction should be aborted and the transaction that started early should be allowed to continue. For each data item x, two timestamps are tracked:

- Max-rts(x): max time stamp of a transaction that read x.
- Max-wts(x): max time stamp of a transaction that wrote x.

- $Read_i(x)$
  - If $ts(T_i) < max\text{-}wts(x)$ then Abort T_i
  - Else
    - Perform $R_i(x)$
    - $Max\text{-}rts(x) = \max(max\text{-}rts(x), ts(T_i))$
- $Write_i(x)$
  - If $ts(T_i) < max\text{-}rts(x)$ or $ts(T_i) < max\text{-}wts(x)$ then Abort T_i
  - Else
    - Perform $W_i(x)$
    - $Max\text{-}wts(x) = ts(T_i)$

Figure 15.11: Read-writes using timestamps

Conflicts are handled using both these timestamps as shown in the Figure 15.11. If a transaction want to perform read opearion on data item x, the last write on that data item is checked. The transaction timestamp is compared with the last write timestamp of the data. If the later transaction modified the data, the transaction is aborted. If the read is successful, the read timestamp is updated by calculating max of the previous timestamp and timestamp of current transaction as shown in the above figure.

In case of a write, if there is any more recent transaction that has read or modified the data item, the transaction is aborted. If the write is performed, the timestamp of data item is updated.

Question: When you undo the transaction, do you undo the changes?
Answer: During abort, the state is restored to the original values using the undo log in case of Write-ahead log and copies are deleted in private workspace model.

Question: How do we ensure the atomicity of read, write operations and the checks made in Figure 15.11?
Answer: A lock is grabbed while performing all of the operations shown in the figure to prevent any other transaction from making changes.

Question: When a transaction is aborted, is it just killed or rerun again?
Answer: There are two ways to handle this. One way is to inform the application that the transaction is

aborted and let the application rerun the transaction again. Another way is to make the transaction manager retry the transaction.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 16: March 28

Lecturer: Prashant Shenoy *Scribe:* ***Shishir Verma(2017), Priyanka Mary Mammen(2022)***

16.1 Consistency and Replication

Replication in distributed systems involves making redundant copies of resources, such as data or processes, while ensuring that all the copies are identical in order to improve reliability, fault-tolerance, and performance of the system.

Types of Replication :

- **Data replication:** When the same data are stored on multiple storage devices.
- **Computation replication:** When the same computing task is available to be executed on multiple servers.

16.1.1 Why Replicate?

- **Reliability:**

  Data in distributed systems need to be replicated to improve the reliability of the system. If one of the replicas become unavailable or crashes, the data still remain available. For instance, in a distributed system, if one of the database servers crashes and we have a replicated copy of the same data on another database server, then the data is safe. We can point our system to the second replica of the database and continue to access the data without any problems. This is in general true with any storage system. If we have multiple copies of the data and the disk crashes on one machine or something else goes wrong, our data remain available because we have other copies.

  In many cloud-based storage systems, like Amazon S3, replication is done internally. It replicates the data in multiple locations. User can ask for a copy of the data and the system will get it from one of the replicas. The user doesn't have to know or specify from which replica the data should be accessed. There are many file-systems that support replication as well, e.g., hadoop file system (hdfs) or Google file system (GFS).

- **Performance:**

  Computation or data are also replicated to improve the performance of the system. Replicated servers can serve a larger number of users as compared to just one server. For example, if we have just one web-server, it would have a certain capacity, i.e., requests it can serve per second. After reaching the limit, it will get saturated. By replicating it on multiple servers, we can increase the capacity of our application so that it can serve more requests per second.

  Similarly, data can also be replicated to improve performance and capacity of the system. For instance, if we have a large number of web-servers and just one database server, eventually, the requests from web-servers will trigger more queries than what the database is capable of executing. If those are computationally expensive queries, the database might become the bottleneck in the system. In ideal

case, you would expect a linear increase in throughput, but in most cases you would get something less than ideal performance.

The replication can also be done in wide area networks, i.e, you can put copies of your applications in different geographical locations (which is called *geo-distributed replication*). Here, we are keeping copies closer to users, which aids better performance due to the decreased latency when accessing the application.

16.1.2 Replication Issues

Before we get into consistency, we will discuss replication issues that we have to consider:

- **When to replicate?**
  Similar to dynamic-or-static threadpool concept.
- **How many replicas to create?**
  If we need to sustain a certain request rate, we can find out how many replicas are required depending on the individual capacity of each replica.
- **Where should the replicas be located?**
  In a distributed application we can put the replicas in different locations. The general rule of thumb is to keep the servers geographically closer to the end-users. If the users are spread out in several locations, then it would be wise to keep replicas spread out in similar fashion. The users can connect to the replica that is geographically closest to them.

16.2 CAP Theorem

The CAP theorem was initially a conjecture by Eric Brewer at the PODC 2000 conference. It was later established as a theorem in 2002 (by Lynch and Gilbert). The CAP theorem states that it is impossible for a distributed system to simultaneously provide more than two out of the following three guarantees:

Consistency (C): A shared and replicated data item appears as a single, up-to-date copy

Availability (A): Updates will always be eventually executed

Partition-tolerance (P): The system is tolerant to the partitioning of a process group (e.g., because of a failing network)

16.2.1 CAP Theorem Examples

Consistency + Availability: Single database, cluster database, LDAP, xFS. If you want to have consistency and availability in your system, you have to assume that network cannot be partitioned to ensure that messages do not get lost.

Consistency + Partition-tolerance: Distributed database, distributed locking. They assume that the coordinator doesn't fail and there wont be any modifications in the system.

Availability + Partition-tolerance: Coda, web caching, DNS. DNS updates can take upto few days to propagate.

16.2.2 NoSQL Systems and CAP

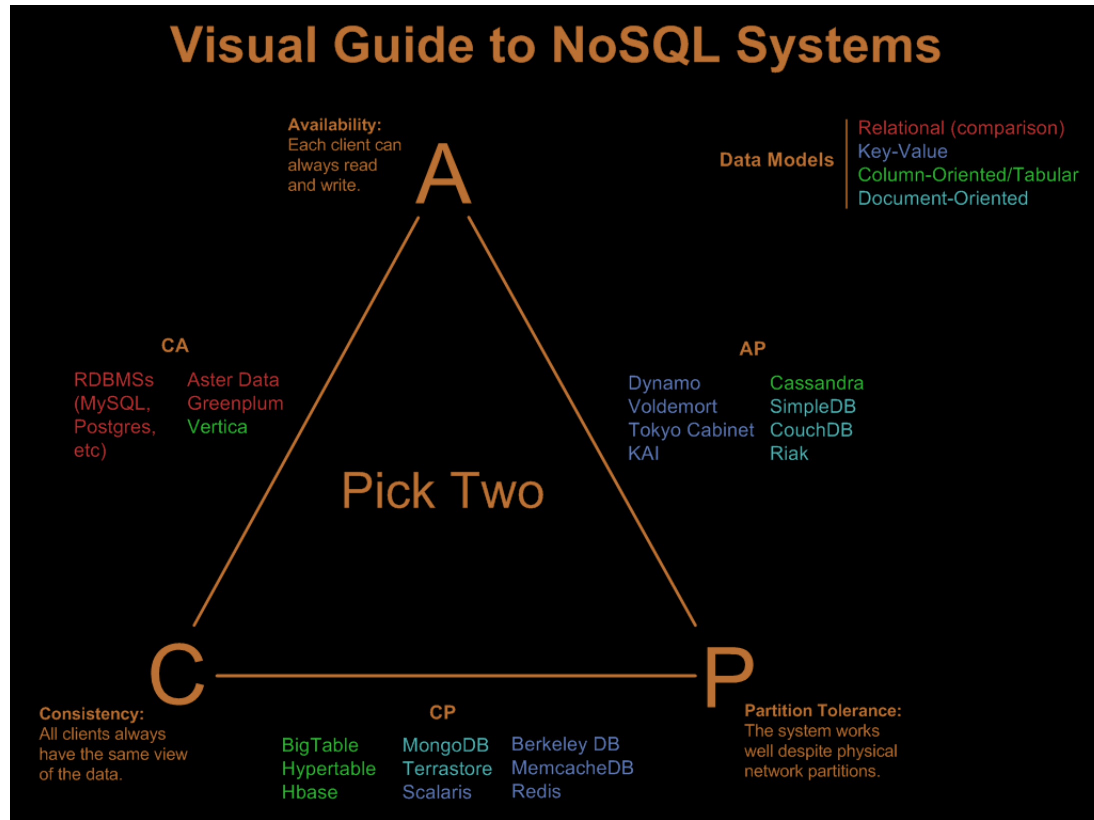


Figure 16.1: CAP in database systems.

NoSQL systems, as the name suggests, don't use a SQL database. Figure 16.1 shows some database systems and which properties they hold. Consistencies in the presence of network partitions are problematic.

Question: What is partition tolerance ?
Answer: Partition tolerance in CAP means tolerance to a network partition. Suppose there are some nodes in a distributed system and they are connected over the Internet. If any of the link goes down, the network essentially is partitioned into two halves. The nodes in first half can talk to one another, and the nodes in second half can talk to one another, but the nodes from first half cannot talk to the nodes in second and there are clients able to talk to either one or both of those nodes. In our case these are replicas. If there is an update on the node, that update can be propagate to other nodes, but since the network is partitioned it cannot communicate with the other nodes until the network is fixed. The system will be inconsistent if the messages are not flowing back and forth.

Question: Why is availability an issues?
Answer: Availability can be an issue if a node goes down and the system can't make any progress. For example, in the case of distributed locks, if a nodes go down, we cannot actually operate our system. Similarly in the case of 2-phase commits and and other situations where it is required for all the nodes to agree on something, if some nodes are unavailable, then they will not be able to agree.

Question: Is there any way we can relax one of the dimensions, e.g., consistency and get more of the other dimensions?
Answer: For specific systems we can make trade-offs. There is no general rule saying that if we relax

property A by 20%, we can get 30% more of property B because it all depends on the assumptions we make for that application.

Question: In Figure 16.1, there are a lots of databases mentioned. Some of them offer availability and partition tolerance, but not consistency. Why would a database not want consistency?
Answer: In these cases it means that we are not getting good consistency guarantees. A very lose form of consistency is called "eventual consistency." The best way to understand it is by taking DNS as an example. We can think of DNS as a very large database that stores hostname to IP address mappings. There are no consistency assumptions made. If we make an update, it may take up to 24 hours for it to propagate. Until then, things may be inconsistent with respect to one another. We do this because we want availability and partition-tolerance. If our application needs a better guarantee than that, we should not choose these databases.

16.3 Object Replication

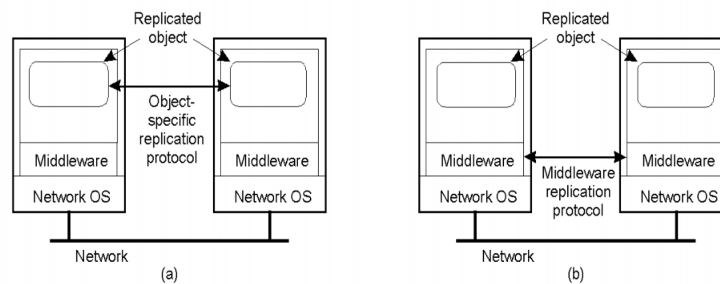


Figure 16.2: Two types of replication. (a) The application does the replication and handles consistency. (b) The middleware does the replication and handles consistency.

Question: Is it beneficial to implement replication in the middleware than in the application level?
Answer: It depends on the type of application you use.

16.3.1 Replication and scaling

Replication and caching are often used to make systems scalable. Suppose an object is replicated N times, the read frequency is R, and the write frequency is W. Stricter consistency guarantees are worthwhile if $R >> W$, otherwise they are just wasted overheads (tight consistency requires globally synchronized clocks). The overheads increase as we make the consistency guarantees stricter. Thus, we try to implement the loosest consistency technique that is suitable for our application.

16.4 Data-Centric Consistency Models

We can analyze from the perspective of data items. Tthere are consistency models from the perspective of clients too. All of the consistency models have the goal to retrieve the most recently modified version. There is a contract between the data-store and processes, i.e, if processes obey certain rules, the data store will work correctly. All models attempt to return the results of the last write for a read operation.

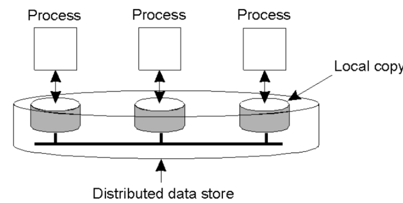


Figure 16.3: Data-centric consistency models.

16.4.1 Strict Consistency

Strict consistency is when the system always returns the results of the most recent write operation. There is no inconsistency. It is hard to implement as it assumes a global clock and compares the read and write timings. There is some delay in propagation of messages as well. Suppose a copy at location A gets modified and A sends a notification to B about its write which takes 1ms to travel. If B gets a read request before the message from A has arrived but after A has been modified, B will not know that there has been an update. This is one of the reasons why it is so hard to implement.

16.4.2 Sequential Consistency

Sequential consistency is weaker than strict consistency. All operations are executed in some sequential order which is agreed upon by the processes. Within a process, the program order is preserved. We can pick up any ordering for operations across different machines.

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)b	R(x)a

(a)

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

Figure 16.4: Sequential consistency.

In Figure 16.4, lets say x is a web page. Process $P1$ writes a to x and process $P2$ writes b to x. Process $P3$ reads x's value as b and then later reads it as a. If we had a global lock, we would know that $P1$ wrote it first and then $P2$. So, once $P3$ sees a it shouldn't see b. But since we do not have synchronized clock, we don't really know if that is what has happened, because $P1$ and $P2$ did not communicate with each other and hence we don't know if they are concurrent events. So processes just agreed that $P1$'s write happened before $P2$'s write. In 16.4 (a) The processes agree on the order that $P2$ wrote before $P1$ and both $P3$ and $P4$ read in that order. Figure 16.4 (b), $P3$ and $P4$ see in different orders and that is not allowed.

Question: Process $P1$ has written to the web page, so why does it not see a before b?
Answer : It will process a before b but the question is when the update b arrives, $P1$ has to decide if that update occur before $P2$. j]Just like in totally ordered multicasting, we will have to wait for all the writes to figure a global ordering and then commit them in that order.

16.4.3 Linearizability

Along with all the properties of sequential consistency, we also have the requirement that if there are two operations x and y across different machines such that time-stamp of x, $TS(x) <$ time-stamp of y, $TS(y)$, then x must precede y in the interleaving. There is an implicit message passing. The reads and writes are done on shared memory buffers and if we read some value from a variable, the write must have happened before. If there are concurrent writes, then their order can not be determined. Linearlizability is stricter than sequential consistency but weaker than strict consistency. Consider the difference between serializability and linearizability: serializability is a property at transaction level, whereas linearizability handles reads and writes on replicated data.

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

Figure 16.5: Linearizability example.

Figure 16.5 shows three processes. Each process writes one variable and reads variables written by the others. Thus, there is an implicit communication here about the ordering. The valid interleaving is shown in Figure 16.6.

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

(a)	(b)	(c)	(d)
x = 1;	x = 1;	y = 1;	y = 1;
print ((y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111

Figure 16.6: Valid interleaving for Figure 16.5 satisfying the property of linearizability.

An invalid ordering would be when after assigning a value to a variable we still print a 0. Another scenario will be if we do not agree to a program order.

16.4.4 Causal Consistency

Causally related writes must be seen by all the processes in the same order. In Figure 16.7 (a), $P2$ read a from x and then wrote b which means that $P1$ wrote before $P2$ and thus, a will be read before b. Process $P3$ does not agree to it and thus is not consistent. For concurrent writes, the processes do not need to

agree upon an interleaving and can read in any order (Figure 16.7 (b)). Causal consistency is weaker than linearizability as the latter fixes an order.

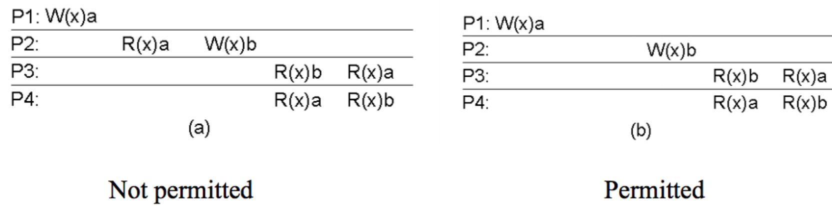


Figure 16.7: Causal consistency.

Question: In Figure 16.7 (b), if $P3$ reads it twice, is it possible for it to read b and b again?
Answer : That is valid because in that case $P1$ writes a on x first. Then $P2$ writes b on x and it overwrote the content written by $P1$. $P3$ reads that subsequently and keeps seeing b as many times as it reads.

Question: Could you give an exmaple for what is allowed in sequential consistency but not in linearizability (Figure 16.7 (b)?
Answer : Processes can agree on some order, say b, a, then $R(x)b$ $R(x)a$ and $R(x)b$ $R(x)a$ is allowed in sequential consistency, which is not allowed in linearizability.

Question: In lineraizability is $R(x)a$ $R(x)b$ and $R(x)a$ $R(x)b$ allowed (Figure 16.7 (b)?
Answer : Yes, in fact that is required. You have to see a followed by b if there is a happen-before relation.

16.4.5 Other Models

FIFO consistency does not care about ordering across processes. Only the program ordering within a process is considered. It may also be sometimes hard. It is also possible to enforce consistency at critical sections, i.e., upon entering or leaving a critical section but not within a critical section. This can be a weak consistency or entry and release consistency. All transactional systems like databases use this kind of consistency. Consistency is done at commit boundaries only.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Figure 16.8: Consistencies (weaker as you go down).

16.5 Client-centric Consistency Models

Consider reads and writes performed by different clients (processes). There are following types:

Monotonic Reads: All reads after a read will return the same or more recent versions. It does not necessarily have to be the most recent.

Monotonic Writes: The writes must be propagated to all replicas in the same order.

Read your writes: A process must be able to see its own changes. For example, if you update your password and log back in after sometime while the changes have not been replicated. But still, the system should not say incorrect password.

Writes follow reads: The writes after read will occur on the same or more recent version of the data.

Question: What is the concept of "you" in the above description?
Answer: "You" means a machine or a process or a user who uses the machine.

16.6 Eventual Consistency

Because of their high costs, many systems do not implement the consistency models described previously. According to eventual consistency, an update will eventually reach all of the replicas; there are no guarantees regarding how long it will take. DNS uses eventual consistency. The only guarantee is that in the absence of any new writes, all the replicas will converge to the most recent version. Write-write conflicts occur in this model because there can be conflicting writes across machines and eventually there will be a conflict when the updates propagate. Source code control systems are also eventually consistent. Some exmaples of systems that use eventual consistency include:

- DNS: Single naming authority per domain. Only naming authority allowed updates (no write-write conflicts).
- NIS: User information database in Unix systems. Only sys-admins update database, users only read data. Only user updates are changes to password
- Cloud storage services such as Dropbox, OneDrive, and iCloud all use eventual consistency.

Question: If DNS uses eventual consistency, can it lead to network problem?
Answer: The problem occurs because many of the DNS servers cache entries. If you want to avoid it, you have to reduce the size of the cache value. In this case, when you make the same request again, that server needs to look upto the origin server again, which might lead to an increase in load. Thus, expiring the cache values will generate more requests at the origin server.

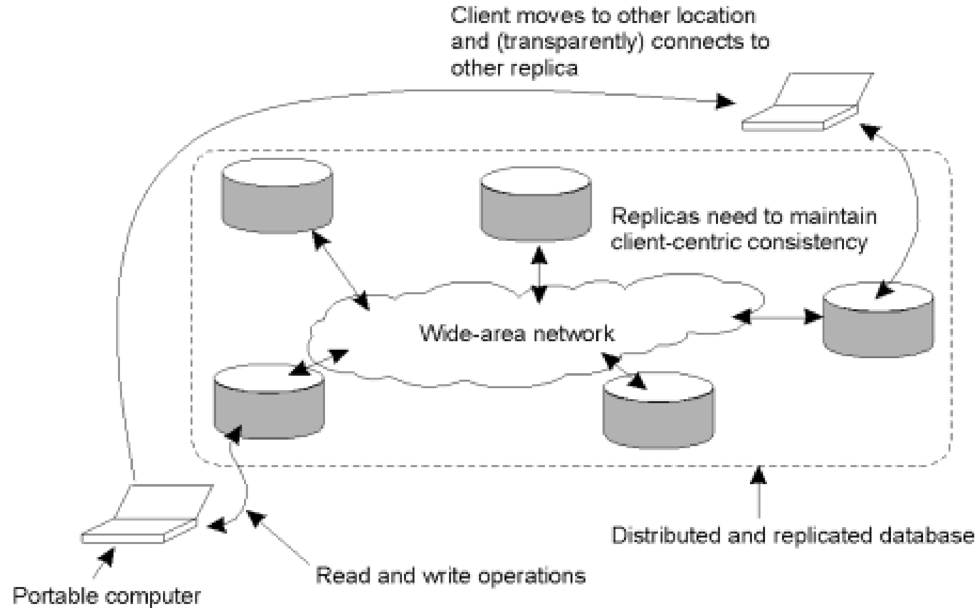


Figure 16.8: Eventual consistency.

16.7 Epidemic Protocols

These protocols help implement eventual consistency. In Bayou, a weakly connected environment is assumed, i.e., clients may disconnect. Offline machines are made consistent when they re-connect (e.g., pulls in git). The updates propagate using pair-wise exchanges similar to diseases. Machines push/pull updates when they connect to another machines and eventually all the machines will have the updates.

Many systems that you encounter in practice, use this form of consistency. For example, DropBox essentially uses this type of model, except that DropBox has a centralized server. You might have many Dropbox clients. When you make an update on one device, your Dropbox client at some point going to contact the centralized server and tell it "Here are some changes," and push it. It might also pull for new updates. Once you pushed your changes to the server, other clients can pull the changes from the centralized server after some time. So you have a pairwise exchange of information between two machines which happens at random intervals.

Question: Will you waste a lot of messages trying to spread an infection? When do you stop?
Answer: There are two algorithms based on epidemic protocols discussed in the following sections and will answer this question there.

16.7.1 Spreading an Epidemic

Algorithms:

- Anti-entropy:
  - Server P picks a server Q at random and exchanges updates.
  - Three possibilities: only push, only pull or both push and pull.

- **Claim**: "A pure push-based approach does not help spread updates quickly."
  Explanation:
  Suppose there is a system with N nodes and we make a change at one of the nodes. This node will randomly pick another node and push that update. Next, these two nodes will pick two other nodes randomly and push the update. The number of nodes which have the update increase exponentially. In the end there will be a very small set of servers which haven't received the update. The probability of picking a server in a large system is $1/N$ i.e. for a large value of N, it is a small probability. We may end up picking up the same servers which have already seen the update. So, the remaining small number of nodes may not get the update quickly. We will have to wait until one of these infected nodes end up picking them and push the update.

It works much better if we combine push and pull because nodes are pro-actively pulling and pushing.

- Rumor Mongering (also known as "gossiping"):

  This works similar to how rumors are spread. Inspired by class of protocols called *gossip protocol*, which are same as epidemic protocols with one small difference: in Rumor mongering there is some probability that you will stop. Just as initially if we have news item, we try to spread it, but after a while we feel like everybody knows it, so we stop calling friends. Rumor mongering is a push-based protocol.

  - Upon receiving an update, P tries to push to Q.
  - If Q already received the update, stop spreading with prob 1/k.
  - Analogous to "hot" gossip items => stop spreading if "cold."
  - Does not guarantee that all replicas receive updates.
    * Chances of staying susceptible: $s = e^{-(k+1)(1-s)}$

Question: Can you push faster and at a higher rate in anti-entropy?
Answer: The rate at which you push or how frequently you push is a parameter you can set in both anti-entropy and rumor mongering, so both of them can control the rate at which spread is happening.

Question: There are many ways to do this, are there any reasons why choose this?
Answer: That's right, this is an entire area of research and there are hundreds of papers published on approaches similar to these.

Question: If a file is changed at location 1 and some other client changes the same file at another location at around the same time, what happens?
Answer: This is called a write-write conflict. This will often occur in systems like Dropbox. If you login on two machines, open the same file on both the machines, make two different changes and save the file more or less at the same time, you will see both of those clients will try to contact the server and server will see that the files are changing more or less at the same time, it will declare a write-write conflict and create two copies, saying that the file changed at the same time. This can often happen because the consistency guarantee is weak.

16.7.2 Removing Data

Deletion of data is hard in epidemic protocols. Lets say we delete a file from Dropbox. Our Dropbox client contacts the server asking for updates. It will compare the two directories and find a file on the server which is not available on the client. If we simply do pairwise exchange blindly, we will recreate the same file on the client which was deleted. There has to be a way to distinguish between an "update" and a "delete." A "delete" that leaves no sign of it will not allow you to figure out whether it is a deleted file or a new file

that got added. This problem is solved using *death certificates*, which means when a file is deleted, an entry is kept for the file that has been deleted. So, “delete” is now an “update,” which has to be propagated and cause other nodes to delete the file as well.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 17: March 30

Lecturer: Prashant Shenoy Scribe: Susmita Madineni (2022), Chen Qu (2019), Sheshera Mysore (2017)

17.1 Overview

This lecture covers the following topics:

1. Primary-based protocols
2. Replicated writer protocols
3. Quorum-based protocols
4. Replica Management
5. Fault tolerance

17.2 Implementing Consistency Models

There are two methods to implement consistency mechanisms:

1. **Primary-based protocols** These work by designating a primary replica for each data item. Different replicas could be primaries for different data items. The updates of a file are always sent to the primary first and the primary tracks the most recent version of the file. Then the primary propagates all updates (writes) to other replicas. Within primary-based protocols, there are two variants:

   Remote write protocols: All writes to a file must be forwarded to a fixed primary server responsible for the file. This primary in turn updates all replicas of the data and sends an acknowledgement to the blocked client process making the write. Since writes are only allowed through a primary the order in which writes were made is straightforward to determine which ensures sequential consistency.

   Local write protocols: A client wishing to write to a replicated file is allowed to do so by migrating the primary copy of a file to a replica server which is local to the client. The client may therefore make many write operations locally while updates to other replicas are carried out asynchronously. There is only one primary at anytime.

   Both these variants of primary-based protocols are implemented in NFS.

2. **Replicated write protocols:** These are also called *leaderless protocols*. In this class of protocols, there is no single primary per file and any replica can be updated by a client. This framework makes it harder to achieve consistency. For the set of writes made by the client it becomes challenging to establish the correct order in which the writes were made. Replicated write protocols are implemented most often by means of quorum-based protocols. These are a class of protocols which guarantee consistency by means of voting between clients. There are two types of replication:

Synchronous replication: When the coordinator server receives a Write request, the server is going to send the request to all other follower servers and waits for replication successful acknowledgements from them. Here, the speed of the replication is limited by the slowest replica in the system.

Asynchronous replication: In this, the write requests are sent to all the replicas, and the leader waits for the majority of the servers to say " replication is successful" before replying to the client that the request is completed. This makes it faster than synchronous replication. But the limitation is when we perform a read request on a subset of servers who has not yet completed the replication, we will get old data.

Question: In primary-based protocols, do we need to broadcast to all clients the fact that the primary has been moved?
Answer: Typically no. But it depends on the system. Ideally, we don't want to let the client know where the primary is. The system deals with it internally.

Question: Do the servers need to know who the primary is?
Answer: Yes.

Question: Do replicated write protocols always need a coordinator?
Answer: It is not necessary to have a coordinator if the client knows what machines are replicated in the system.

Question: How do we prevent cleitns from performing reads on a subset of servers who have not completed the replication yet?
Answer: From a consistency standpoint, asynchronous replication violates read-your-writes.

17.3 Quorum-based Protocols

The idea in quorum-based protocols is for a client wishing to perform a read or a write to acquire the permission of multiple servers for either of those operations. In a system with N replicas of a file if a client wishes to read a file it can only read a file if N_R (the read quorum) of these replica nodes agree on the version of the file (in case of disagreement a different quorum must be assembled and a re-attempt must be made). To write a file, the client must do so by writing to at-least N_W (the write quorum) replicas. The system of voting can ensure consistency if the following constraints on the read and write quorums are established:

$$N_R + N_W > N \tag{17.1}$$

$$N_W > N/2 \tag{17.2}$$

This set of constraints ensures that one write quorum node is always present in the read quorum. Therefore a read request would never be made to a subset of servers and yield the older version file since the one node common to both would disagree on the file version. The second constraint also makes sure that there is only one ongoing write made to a file at any given time. Different values of N_R and N_W are illustrated in Figure 17.1.

Question: Can you actually require N_R to be less than N_W?
Answer: Yes, else we will always pick one server that is never in the Write Quorum

Question: Will you check different combinations of N_R servers to get a successful read?
Answer: Consider $N_R = 3$. Pick 3 random servers and compare their version numbers as part of the voting

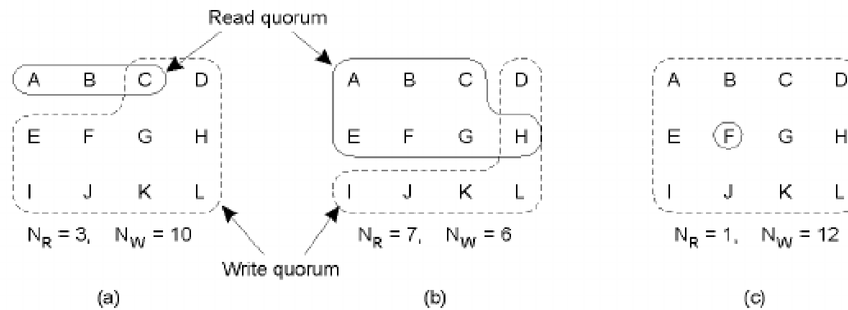


Figure 17.1: Different settings of N_R and N_W.

phase. If they agree, then that is going to be the most recent version present and read is successful, otherwise the process needs to be repeated. If the number of servers is large and the write quorum is small, then you have to have multiple retries before you succeed. To avoid this make the write quorum as large as possible. If the write quorum is large, there is a high chance that read quorums will succeed faster.

Question: Why do we need them to agree on the version if we just do some reads and pick the one with higher quorum?
Answer: Consider that version of file is four in the servers. Consider that you have performed two writes. In the first case, servers A, B, C, E, F, and G are picked and they update the version number to five. In the second case servers D, H, I, J, K, and L are picked and they have also updated the version number to five. In these scenarios, we will have a write-write conflict.

Question: Should all write quorum nodes be up to date before a new write is made?
Answer: Here we assume that a write re-writes the entire file. In case parts of the file were being updated this would be necessary.

Question: Should all writes happen atomically?
Answer: This is an implementation detail, but yes. All the writes must acknowledge with the client before the write is committed.

Question: Can you read from the read quorum and just select the maximum version file? *Answer*: Yes. But you want them to agree.

17.4 Replica Management

Some of the design choices involved in deciding how to replicate resources are:

- Is the degree of replication fixed or variable?
- How many copies do we want? The degree of three can give reasonable guarantees. This degree depends on what we want to achieve.
- Where should we place the replicas? You want to place replicated resources closer to users.
- Can you cache content instead of replicating entire resources?

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 17.2: Failure tyles.

- Should replication be client-initiated or server-initiated?

Question: Is caching a form of client-initiated replication?
Answer: Yes, but client-initiated could be broader than just caching of content, it could even be replication of computation. In case of gaming applications client demand for the game in a certain location may lead to the addition of servers closer to the clients. This would be client initiated replication as well.

17.5 Fault Tolerance

Fault tolerance refers to ability of systems to work in spite of system failures. This is important in large distributed since a larger number of components implies a larger number of failures, which means the probability of at least one failure is high. Failures themselves could be hardware crashes or software bugs (which exist because most software systems are shipped when they are "good enough").

One perspective on fault tolerance is that computing systems are not very reliable. Some computing systems are mission critical, such as auto-pilot on a car or smart TV. We cannot/don't want to simply reboot these systems when they fail, so fault tolerance is important. In addition, we need to make computing systems more reliable.

A system's *dependability* is evaluated based on:

- Availability: The percentage of time for which a system is available. Gold standard is that of the "five nines" i.e a system is available 99.999% of the time. This translates to a few minutes of down-time per year.
- Reliability: System must run continuously without failure.
- Safety: System failures should not compromise safety of client systems and lead to catastrophic failures.
- Maintainability: Systems failures should be easy to fix.

There are many types of faults, including:

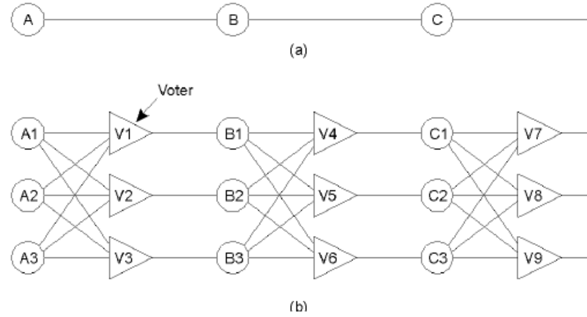


Figure 17.3: Failure masking by redundancy

- Transient faults: When the system is running, it sees occasional errors but continues to run. The errors come and go, but do not bring the system down.
- Intermittent faults: The system may die occasionally but if you restart it, it comes back up.
- Permanent faults: the system is dead and not coming back up.

The different models of failure are shown in Figure 17.2. Typically fault tolerance mechanisms are assumed to provide safety against crash failures. Arbitrary failures may also be thought to be Byzantine failures where different behavior is observed at different times. These faults are typically very expensive to provide tolerance against.

17.5.1 Redundancy

Fault tolerance may be achieved by means of redundant computations and per stage voting. The circuit shown in Figure 17.3 demonstrates this. Here each computation of the stages A, B and C is replicated and the results are aggregated by votes. This circuit is capable of tolerating one failure per stage of computation. If we try to deal with crash fault, we only need the replication degree to be 2, because we assume the node always produces the correct result if it's alive. We need the replication degree of 3 to deal with Byzantine fault.

Question: Why replicate the voter?
Answer: Voters can fail. Replicate the voter makes the system more resilient.

Question: What if V1 fails and then B2 fails?
Answer: If V1 fails, it is not going to get a result and if B1 fails it's going to produce a bad result. The system can tolerate one failure at a time. It cannot handle a voter and the next component in conjunction with to fail. Parallel failures in multiple stages cannot be handled.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 18: April 4

Lecturer: Prashant Shenoy *Scribe:* ***Samika***

18.1 Agreement in Faulty Systems

The two main type of faults are crash failures and Byzantine faults. Fault tolerance during crash failures allows us to deal with servers which crash silently. Detecting failures can be achieved by sending "heartbeat" messages. In a system where we only have silent faults, if the system has k faults simultaneously then we need $k+1$ nodes in total to reach agreement. In Byzantine faults, the server may produce arbitrary responses at arbitrary times. It needs higher degrees of replication to deal with these faults. To detect k byzantine faults, we need $2k+1$ processes.

18.2 Byzantine Generals Problem

In the Byzantine generals problem, we have a scenario in which two generals in spatially separated camps want to reach a consensus on whether to attack a fort. The attack will only be successful if both generals attack. They send messengers in order to communicate with each other. Each general sends their vote message containing "attack" or "retreat," and the other general sends an "ack"to the vote. However, the messengers are sometimes killed by the enemy before delivering the message. Therefore, the communication channel is unreliable and both generals do not know if their votes or acks were reliably received. In this setup, it is provably impossible for the generals to reach a consensus.

For example, suppose the first general sends an "attack" message. The second general recieves the message and sends and "ack," but the messenger carrying the "ack" is killed. The first general never receives the ack, so they do not attack. The second general attacks, but the attack fails because the first general does not attack. Suppose instead the second general waits to receive an additional "ack" from the first general before attacking. But suppose the third messenger carrying the second "ack" is killed—now the first general will attack and the second general will not. So instead the first general waits for the second general to send a third "ack"... and so on, ad infinitum.

Question: Can you reach probabilistic agreement in the Byzantine general problem?

Answer: The network of communication in this problem is faulty, thus there is no such thing as probabilistic agreement. This problem cannot be solved if the underlying communication network is faulty.

If the communication network is reliable, thus there is a solution to reach consensus. Byzantine faults can be modeled as a consensus problem (Byzantine generals problem) among nodes in presence of faulty processes assuming that a Byzantine node will force the system to not reach consensus. A recursive solution to the problem is provided in Figure 18.1. In this, each node collects information from all other nodes and sends it back to all others so that each node now can see the view of the world from the perspective of other nodes. By simple voting, each node can now either accept a single correct value or can identify a Byzantine failure. In a system with k such faults, $2k+1$ total nodes are needed to only detect that fault is present, while $3k+1$ total nodes are needed to reach agreement, despite the faults.

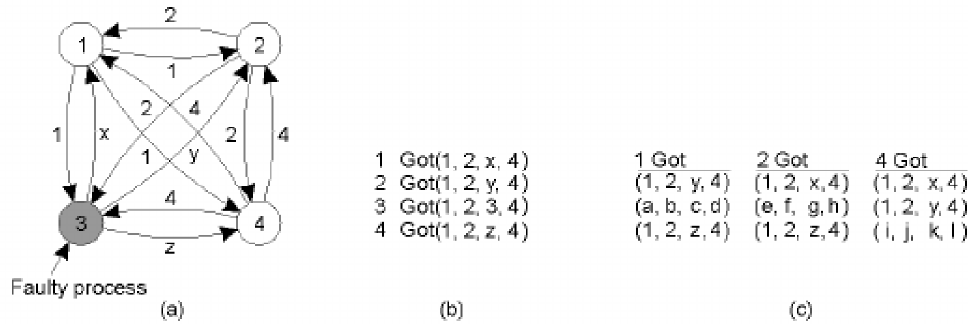


Figure 18.1: Solution to the byzantine general's problem over a reliable communication network.

Byzantine faults need a higher degree of replication as stated above to be solved. Thus, handling these kind of faults is expensive.

18.3 Reaching Agreement

If message delivery is unbounded, no agreement can be reached even if one process fails and slow processes are indistinguishable from a faulty ones. If the processes are faulty, then appropriate fault models can be used such as BAR fault tolerance where nodes can be Byzantine, altruistic, and rational.

18.3.1 Reliable One-To-One Communication

One-one communication involves communication between a client process and a server process whose semantics we have already discussed during RPCs, RMIs, etc. In this we only discussed one-to-one communication, but here we are discussing replication. We need one-to-many communication (multicast or broadcast) in order to reach agreement. We need to extend the one-to-one scenario to the many-to-one scenario in order to solve the agreement problem. Figure 18.2 depicts several failure modes in the one-to-one scenario. These failures can be dealt by (1) Using reliable transport protocols such as TCP (b and d can be dealt with in this manner), or (2) handling failures at the application layer. (a, c and e can be dealt with in this manner)

18.3.2 Reliable One-To-Many Communication

If there are lost messages due to network inconsistencies, we need to retransmit messages after a timeout. There are two ways to do this: ACK-based schemes and NACK-based schemes.

ACK-based schemes :

- Send acknowledgement(ACK) for each of the message received. If the sender does not receive the ACK from a receiver, after timeout it retransmits the message.
- Sender becomes a bottleneck: ACK based scheme does not scale well. As number of receivers in the multicast group grows (say 1000 - 10,000) then the number of ACK messages that needs to be processed also grows.

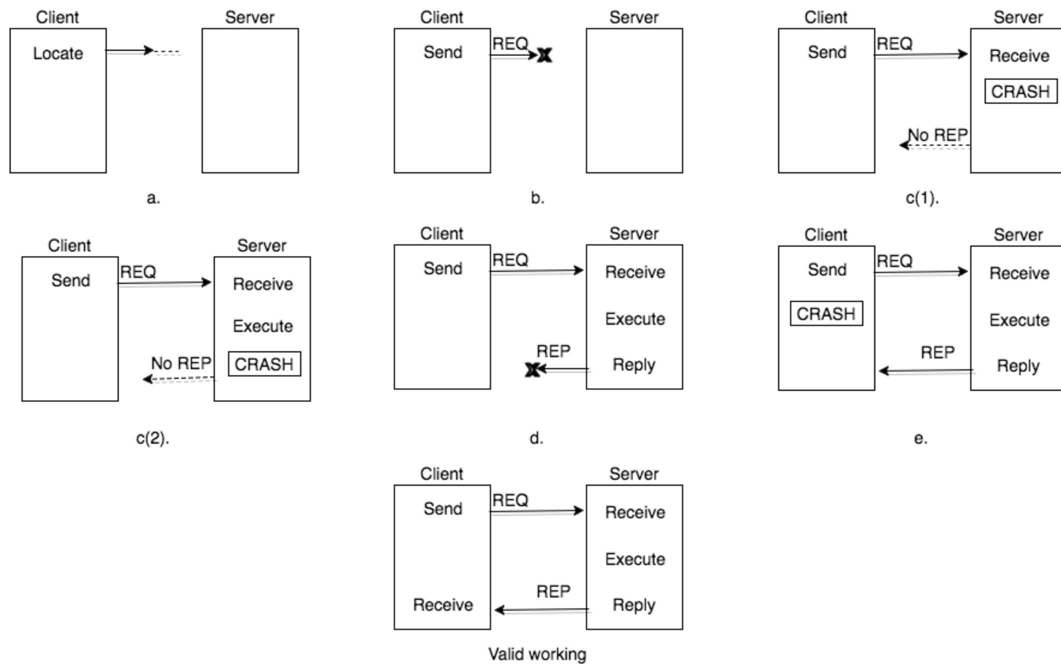


Figure 18.2: Types of failures in the one-to-one scenario. (a) Client unable to locate server. (b) Lost request messages. (c) Server crashes after receiving request. (d) Lost reply messages. (e) Client crashes after sending request.

- ACK based retransmission works well for one-one communication but doesnot scale for one-many communication. Large bandwidth gets used in acknowledgment process which results in an **ACK explosion**.

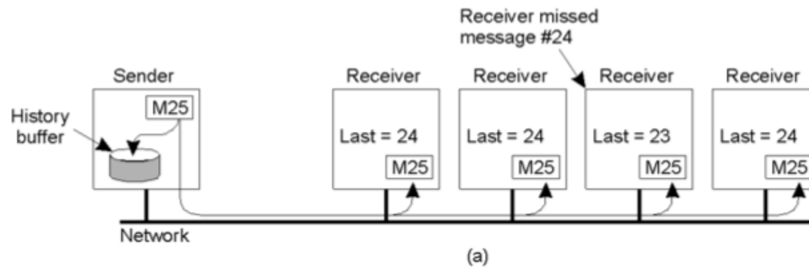


Figure 18.3: Here, all the receivers have their last packet received as #24 except receiver 3 which missed packet #24. Hence, it's last packet is #23. As soon as it receives packet #25, it knows it missed the packet #24.

Question How to reduce the overhead of ACK in one-many communication?
Ans. Instead of sending acknowledgements send negative acknowledgement (NACK).

NACK-based schemes :

- NACK-based schemes deals with sender becoming a bottleneck and the ACK-explosion issue.
- ACK indicates a packet was received. NACK indicates a packet was missed.

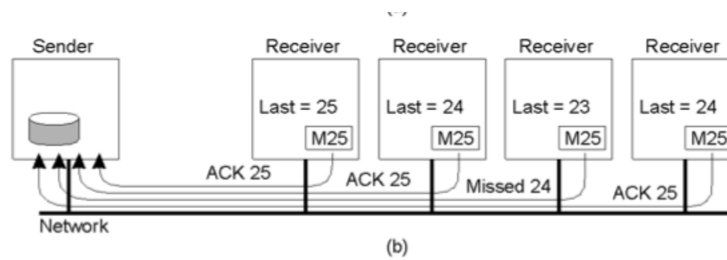


Figure 18.4: Each receiver now sends an acknowledgment ACK either in form of received packet #25 or missed packet #24. As we can see for a single packet, sender receives 'n' ACKs

- Scheme explanation: Send packet to multicast group, if receivers receives a packet, they don't do anything. If receiver sees a missing packet, it sends a NACK to nearby receiver as well as the sender. Sender or neighbouring receivers would re-transmit the missed packet. This optimization works only if the neighboring receivers have the received packets stored in a buffer.
- Sender receive only complaint about the missed packets and this scheme scales well for multicast as the #NACKs received is far less than the #ACKs, unless a massive amount of packet loss.

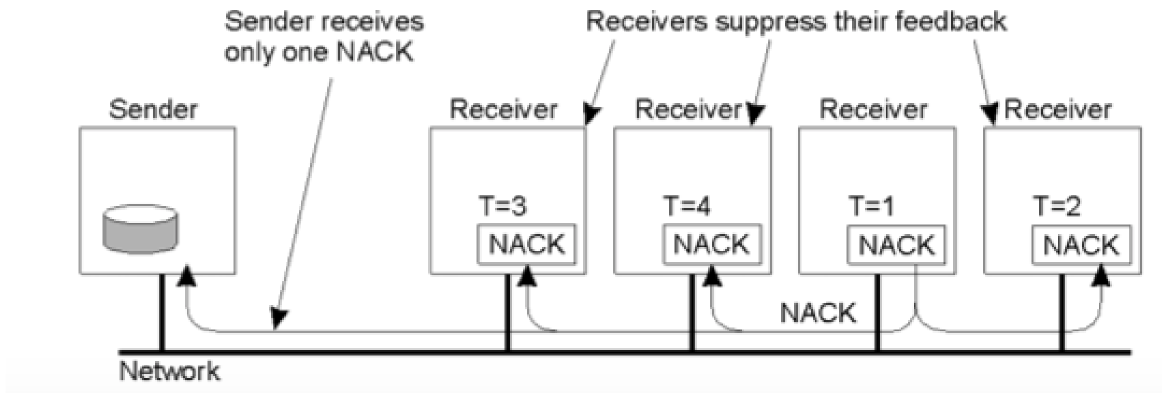


Figure 18.5: Each receiver now suppresses their ACK feedback. Only receiver 3 sends a NACK to other receivers and the sender.

Question How does the receiver know that it missed a packet?
Ans. Assuming the packets are received in sequence and each packet have a packet number. If receiver sees a gap in the sequence, it knows a packet was missed and sends a NACK.

Question Is their a possibility that receive can move from one IP address to another ?
Ans. This possibility exists and is true if its one-one or one-many communication. Socket connection breaks if the IP address changes and connection needs to be re-established. The above mentioned schemes does not handle node mobility.

Question How to deal with last packet or if sender sends only one packet as receiver may never know if it missed the packet?
Ans. Send Dummy packet at the end of transmission and to make sure that dummy packet is acknowledged.

This scheme, only addresses how to send a message to all members of the group, it does not discuss other properties of multicast like:

FIFO order: Messages will be delivered in the same order that they are sent.

Total order: All processes receive messages in the same order. Total order does not require FIFO.

Causal order: It is based on the happens before relationship. If send(m1) happens before send(m2), then the receive(m1) should also happen before receive(m2) between processes.

18.3.3 Atomic multicast

Atomic multicast guarantees **all or none**. It guarantees that either all processes in a group receive a packet or no process receives a packet.

Replicated databases We can't have a scenario where M out of N DB replicas have executed some DB update an the rest haven't. It needs to be ensured that every update to the database is made by all or none.

Problem How to handle process crashes in a multicast?

Solution Group view: Each message is uniquely associated with a group of processes.

If there is a crash:

- Either every process blocks because 'all' constraint will not be satisfied.
- Or all remaining members need to agree to a group change. The process that crashed is ejected from the group.
- If the process rejoins, it has to run techniques to re-synchronize with the group such that it is in a consistent state.

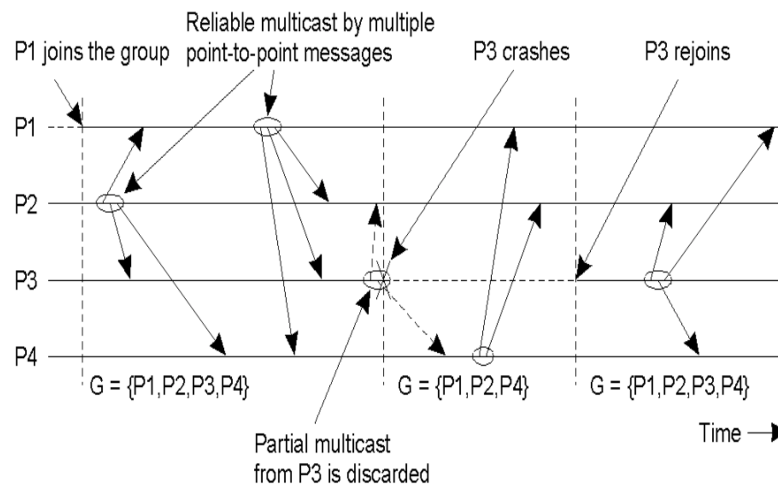


Figure 18.6: Initially all process are up and are part of a group {P1,P2,P3,P4}. All the messages are being reliable multicasted to each of the processes. At dotted line2, P3 crashes while sending a message. From this point onwards, the group {P1,P2,P3,P4} will not maintain the 'all' property of atomic multicast. Hence, P1, P2 and P4 agree on a group change and then start atomic multicast amongst themselves (the new group). At a later point P3 recovers and rejoins. At this point, it run synchronization algorithms to bring itself up-to-date with other members of the group it wants to rejoin.

18.3.4 Implementing virtual synchrony

Reliable multicast and atomic multicast are only two ways of implementing virtual synchrony. There are many variants of these techniques as well as other virtual synchrony techniques which may be used in different application based on the requirements of the application.

- Reliable multicast: Deals only with network issues like lost packets or messages. There is no message ordering. NACK based.
- FIFO multicast: Variant of reliable multicast where each sender's message are sent in order. But, there is not guarantee that messages across senders would be ordered as well.
- Causal multicast: Variant of reliable multicast. Causal dependence across messages which are sent in order.
- Atomic multicast: Totally ordered, all or nothing delivery. Deals with process crashes
- FIFO atomic multicast: Variant of atomic multicast.
- Causal atomice multicast: Variant of atomic multicast.

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Figure 18.7:

Question: Why does atomic multicast have total ordering ?

Answer: Atomicity is a stronger property than total order, it is more expensive. Thus the system may as well have total order if it has atomicity.

Question: What happens in the scenario where some processes receive the message and other do not receive the message ?

Answer: There is a difference between receiving or delivering a message and applying/comitting the message. The commit of a message should take place only after consensus in order to ensure safety, this is discussed later on in the lecture.

Question: Is causal ordering stricter than FIFO?

Answer: Yes, FIFO only ensures ordering within a process whereas, causal ordering ensures ordering across processes.

18.4 Distributed commit

Atomic multicast is an example of a more general problem where all processes in a group perform an operation or not at all. Examples:

- Reliable multicast: Operation = Delivery of a message
- Atomic multicast: Operation = Delivery of a message
- Distributed transaction: Operation = Commit transaction

Possible approaches

- Two phase commit (2PC)
- Three phase commit (3PC)

18.4.1 Two phase commit

Two phase commit is a distributed commit approach used in database systems which takes into account the agreement of all the processes in a group which have replicated database copies. This approach uses a coordinator and has two phases:

- Voting phase: Processes vote on whether to commit
- Decision phase: Actually commit or abort based on the previous voting phase

The algorithm for this approach can be explained using Fig 18.7.

- The coordinator first prepares or asks all the processes to vote if they want to abort or commit a transaction.
- All the processes vote. If they vote commit, they are ready to listen to the voting results.
- The coordinator collects all replies.
- If all the votes are to commit the transaction, the coordinator asks all processes to commit.
- All processes acknowledge the commit
- In case of even a single abort transaction vote including coordinator process's own abort vote, the coordinator asks all processes to abort.

Question How the coordinator is chosen?
Ans. Leader Election.

Question If the process is voting for aborting, is the process up/down?
Ans. If the process is voting the assumption is that the process is up. If the process is down then there will not be any response. This scheme provides safety property but not liveness property. Drawback of two phase commit process is blocking when the coordinator crashes. If the process crashes, eventually the transaction aborts when the coordinator does not hear back from the process.

Question What if it takes long for the process to vote commit?
Ans. Process can vote to commit and coordinator makes decision to abort or to commit.

Question What if the process is byzantine faulty?
Ans. Two phase commit scheme does not work if the process is byzantine faulty. we are assuming crash fault tolerance in both two phase and three phase commit.

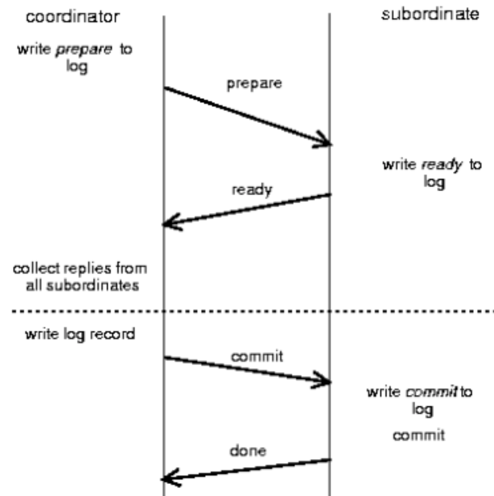


Figure 18.8: Steps showing a successful global commit using 2PC approach

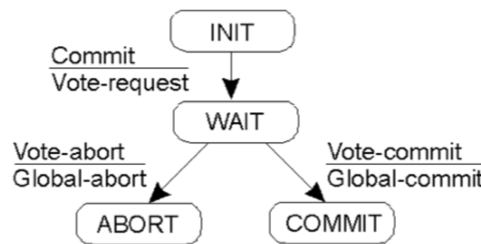


Figure 18.9: **2PC: Coordinator's state transition.** From INIT state, the coordinator asks all processes to vote and goes into WAIT state If any one process votes abort, the coordinator goes to ABORT state and issues global-abort. If all processes vote commit, coordinator goes in COMMIT and issues a global-commit.

Question Is the global abort message sent by coordinator?
Ans. The result of the vote is always sent by the coordinator in decision phase.

Question When the global abort message is sent by coordinator?
Ans. If any process vote abort the coordinator sends global abort to all processes.

Recovering from a crash : When a process recovers from a crash, it may be in one of the following states:

- INIT: If the process recovers and is in INIT state, then abort locally and inform coordinator. This is safe to do since this process had not voted yet and hence coordinator would be waiting for its vote anyway.
- ABORT: The process being in ABORT state means that coordinator would have issued a global-abort based on the abort vote of this process, hence the process can safely stay in the state it is or move to INIT state.
- COMMIT: The process being in COMMIT state means the coordinator already had issued global commit and this process now can safely stay in this state or move to INIT state.

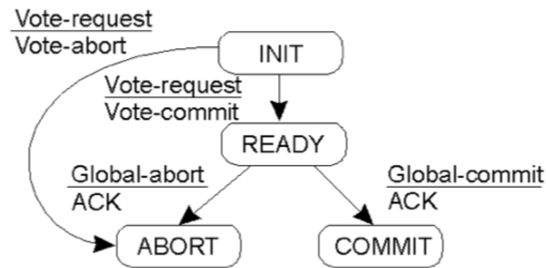


Figure 18.10: **2PC: Subordinate process's state transition.** A process may vote abort and go directly into ABORT state. This is because this single abort would lead to global-abort. A process may vote commit and go into READY state. On being READY and receiving an abort, the process goes into ABORT state. On being READY and receiving a commit, the process goes into COMMIT state.

- READY: The process in this state may be due to a variety of possibilities hence as soon as any process recovers and finds itself in a READY state, it checks other processes for their state to get hint of the group status.
  The table describes the actions of recovered process P on seeing the state of a process Q and the reason for such action.

State of Q	Action by P	Reason
COMMIT	Make transition to COMMIT	Any process can be in commit only if coordinator issued a global-commit
ABORT	Make transition to ABORT	2 scenarios: • If process Q has aborted itself. Then coordinator would issue a global-abort. Hence, P can abort. • If process Q aborted because of a global-abort. P can abort in this case too. .
INIT	Make transition to ABORT	If process Q is in INIT means it has not voted yet. Thus, voting phase is still going on. Process P can abort safely.
READY	Contact another participant	Since, based on process Q's READY state, process P can't infer much. Hence, P should ask another process.

If process Q is in READY : Process Q being in READY state requires a further analysis of action:

- Keep asking other processes about their state
- If at least one of them is not in the READY state then choose an appropriate action from the table above.
- If all of them are in the READY state and are waiting to hear from the coordinator, process P can't make a decision yet. All other processes can't make any decision either.
  The reason: Coordinator itself is a participant in the vote, hence, based on the action it takes after recovering, the option decided by the processes as a group may be wrong. That is:
  - All processes can't just commit because coordinator may recover and want to abort.

– All processes can't just abort because coordinator may recover and see that every process had voted commit and want to commit and issue a global-commit. Other processes in abort state would lead to inconsistent state.

Problem of 2PC If the coordinator crashes without delivering the results of a vote, all processes will be deadlocked. This is called **blocking property of 2 phase commit**.

Question: If the co ordinator has send messages to some processes and not all and then it crashes then what happens ?

Answer: Two properties need to be discussed to **Answer** this, safety and livenss. Safety: Nothing bad happens, the protocol does not reach an incorrect decision. Liveness: There is progress, the protocol reaches a decision. The 2pc guarantees safety and not liveness.

18.4.2 Three phase commit

Three phase commit is a variant of two phase commit which takes care of the liveness property that the 2pc could not guarantee in case of coordinator crash.

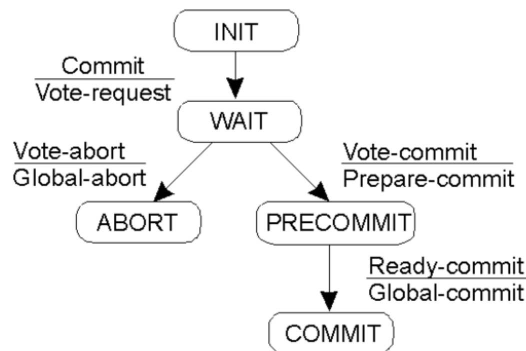
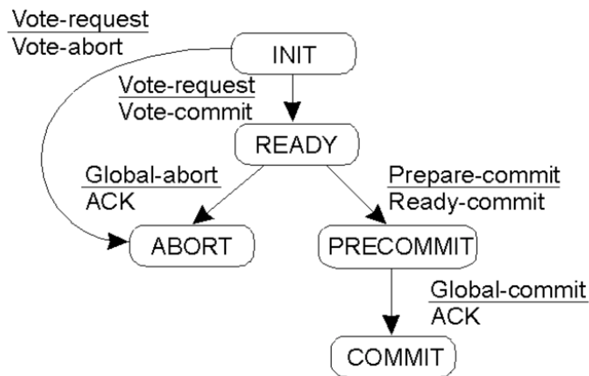


Figure 18.11: **3PC: Coordinator's state transition.** From INIT state, the coordinator asks all processes to vote and goes into WAIT If any process votes abort, the coordinator goes in ABORT and issues global-abort. If all processes vote commit, inform processes to prepare for commit. Go in the PRECOMMIT state. Once all the processes have moved to PRECOMMIT, then issue a global-commit and go into COMMIT state.

How does the 3rd phase PRECOMMIT help? :
Recollecting, blocking problem of 2-phase commit scenario. If a process recovers from a crash and finds itself to be in a READY state, it asks another process about its state. To this the reply is READY state. The processes are still to hear from the coordinator. If every process is in the ready state and the coordinator crashed and can't tell what the outcome of the vote is. A decision can't be made in case of 2PC. However, even in such a scenario a decision can be made safely in case of 3PC. Assuming the coordinator had gone into a PRECOMMIT state and crashed.
The processes can decide among themselves and ABORT.

- If the co-ordinator recovers and finds itself in the PRECOMMIT state, it could ABORT the transaction.



3PC: Subordinate process's state transition. A process may vote commit and go into READY state. It may vote abort and go directly into ABORT state. This is because this single abort would lead to global-abort. On being READY and receiving an abort, the process goes into ABORT state. On being READY and receiving a prepare-commit, the process goes into PRCOMMIT state. Once in PRECOMMIT, the processes move to COMMIT state on receiving global-commit.

- In 2 phase this could not have been possible because the co-ordinator would have gone into COMMIT phase and rest of the processes would have ABORTed leading to an inconsistent state.

Question What happens if co-ordinator crashes after everyone is in pre-commit?
Ans. If every process is in PRECOMMIT and not in READY state, they can go ahead and COMMIT. This is because, once co-ordinator recovers, it may ask other processes for the state to which the reply would be COMMIT. This way co-ordinator can go in a COMMIT state as well.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 19: April 3

Lecturer: Prashant Shenoy *Scribe:* ***Justin Svegliato (2019), Devyani Varma(2022)***

Lecture Overview

- Part 1: Consensus
- Part 2: PAXOS
- Part 3 : RAFT

19.1 Consensus

Definition: get a group of processes to agree on something even when some of the processes fail. More formally, we want to achieve reliability in presence of faulty processes:

- Requires processes to agree on data value needed for computation.
- **Examples**: whether to commit a transaction, agree on identity of a leader, atomic broadcasts, distributed locks.

The failures are in context of crash faults or Fail-stop failures i.e., a process produces correct output while it is running, but the process can hang/go-down and hence will not produce any results. **Note**: When there are no failures, there are protocols like 2 Phase commits that we discussed, to come to an agreement. For instance, committing a transaction.

Byzantine Consensus vs Consensus Byzantine consensus is when we need to come to an agreement in case of processes that are byzantine-faulty i.e., faulty processes continue to run and produce malicious outputs and prevent agreement. Whereas consensus is a benign scenario where some processes fail to respond.

Q: How to decide what consensus protocol to use? Depends on what we're trying to achieve. PAXOS, RAFT can be used for crash faults, for implementing a basic fault tolerance mechanism. Byzantine is more elaborate and used in case where we do not want malicious actors to cause confusion as in case of cryptocurrency.

19.1.1 Properties of a Consensus Protocol

- **Agreement**: Every correct process *agrees* on the same value.
- **Termination**: Every correct process *decides* on some value.
- **Validity**: If all processes *propose* a value(v), all correct processes must *decide* on that value, v.

- **Integrity**:
  - Every correct process *decides* at most one value.
  - If a correct process *decides* on a value, a process must have *proposed* that value.

Q: What does 'all' in validity mean? We will have failures, if any process or say the co-ordinator crashes during agreement, we will not have consensus. But the protocols define that, if we have a majority(and not all) of the nodes up and running and they agree on a value, we have consensus.

19.1.2 2PC/3PC Problems

Both two phase commits and three phase commits experience problems in the presence of different types of failures. While the **safety** property can be ensured, the **liveness** properties cannot always be guaranteed due to node failures and network failurs: the system will never perform an operation that leads to an inconsistent state (satisfying the safety property) but can still be deadlocked (violating the liveness property). We describe a few caveats associated with each type of commit below.

Two Phase Commit

- It must wait for the coordinator and the subordinates to be running.
- It requires all nodes to vote.
- It requires the coordinator to always be running.

Three Phase Commit

- It can handle coordinator failures.
- But network failures are still a problem.

There has been an implicit assumption that there could only be node failures instead of network failures during a two or three phase commit. While a node could crash in the network, the network would never experience any issues. Suppose, however, that the network was partitioned into two partitions due to some problem. Although both partitions will continue to function correctly, each partition cannot communicate with each other. **By definition, if the network is partitioned due to some problem, a two or three phase commit cannot work because every node is required to vote on the answer.**

In order to eliminate such an assumption, we have to revisit the definition of *agreement*. Rather than requiring the vote of every node, we can just require the vote of the majority of nodes. Therefore, if the network were to be separated into two partitions, the partition with the majority of nodes can still continue to function properly. This idea forms the basis of **Paxos**, a consensus protocol. **Instead of requiring every node to vote, Paxos only requires the majority of nodes to vote.**

19.2 Paxos: Fault-tolerant agreement

Paxos lets nodes agree on the same value despite node failures, network failures and network delays. **Use-cases include:**

- Nodes agree X is primary (or leader)
- Nodes agree Y is last operation (order operations)

The protocol is widely used in real systems such as Zookeeper, Chubby and Spanner. ***Leader*** is a process that tries to get other processes to agree on a value. For instance, a process says, I propose that the value after computation is X and gets other process to agree that the output after computation is X. Therefore, leader is essentially a proposer. If majority of the processes agree then, the value is agreed upon. If not, then either the leader tries again or some other process becomes a leader and attempts consensus. **Note:** There can be multiple leaders and can attempt to get others to agree on a value.

19.2.1 Paxos Requirements

Paxos satisfies the following properties:

- Safety (*Correctness*)
  - All nodes must agree on the same value.
  - The agreed upon value must be computed by some node.
  - **Note:** We do not want just trivial consistency i.e.; everyone agrees value is zero or null. Therefore, the value that is agreed upon must be computed by some node.
- Liveness (*Fault Tolerance*)
  - If less than $\frac{n}{2}$ nodes fail, the remaining nodes will eventually reach agreement. This allows the system to make progress in the presence of failures.
  - Note that that liveness is not guaranteed if there is a steady stream of failures as the protocol determines what to do. If a node fails in the middle of the protocol, it must be restarted.
- **Why is agreement hard?** Because even in the face of failures, we still need to reach agreement.
  - The network might be partitioned.
  - The leader may crash during solicitation or before announcing the outcome of voting. While the current round will not produce any results, a new leader will be elected through leader election. All nodes will then vote again.
  - A new leader may propose different values from the value that had been agreed upon originally.
  - Several nodes may become a leader at the same time. This is possible when the network is partitioned due to a network failure. The left half will elect a new leader while the right half will have the old leader, and they will still continue to function properly. Both sides of the partition may agree on different things unfortunately.

19.2.2 Paxos Setup

- Entities: Proposer(leader), acceptor, learner:
  - *Leader* proposes value, solicits acceptance from acceptors.
  - *Acceptors* are nodes that want to agree; announce chosen value to learners
  - *Learners* do not play an active role, but agree on proposed value.

- Proposals are ordered by unique proposal numbers.
  - Node can choose any high number to try and get proposal accepted
  - An acceptor can accept multiple proposals.
    - If a proposal with value v is chosen, all higher proposals have value v.
- Each node maintains:
  - **n_a, v_a:** The highest proposal number and accepted value during that proposal.
  - **n_h:** The highest proposal number seen so far
  - **my_n:** the current proposal number that is in progress.

19.2.3 Paxos Operation : 3 Phase protocol

Phase 1: Prepare Phase Leader understands what other processes have seen or accepted before.

- A node decides to be leader and proposes a value
- Leader chooses my_n >n_h
- Leader sends **<prepare, my_n>** to all nodes. **Note that,** during this, the value proposed is not sent, it's just the prepare message with proposal number.
- Upon receiving **<prepare, n>** at acceptor:
  - If n <n_h: Reply with <**prepare-reject**. (Since, already seen a higher # proposal.)
  - Else:
    - n_h = n (Protocol will not accept proposal lower than n)
    - Reply **<prepare-ok, n_a, v_a >.** (Send back the most recently accepted proposal # and value)
    - Reply can be null, if you haven't seen any proposals yet and this is the first proposal.

Phase 2: Accept Phase

- If leader gets **<prepare-ok>** from majority *(Actions taken by leader)*
  - V = non empty value from the highest n_a received from prepare phase.
  - If V = null, leader can pick any V
  - Send <accept, my_n, V >to all nodes
- If leader fails to get majority **prepare-ok** : Delay and restart paxos.
- Upon receiving <accept, n, V>*(Actions taken by acceptor)*:
  - If n <n_h : Reply with **<accept-reject >**
  - Else : n_a = n; v_a = V, n_h = h; reply **<accept-ok >**

Phase 3: Decide

- If leader gets **<accept-ok >** from majority: Send **<decide, v_a >** to all learners.
- If leader fails to get **<accept-ok >** from a majority: Delay and restart Paxos.

Q: Can Proposals go on indefinitely? At the beginning, no one has agreed to anything, leader gets null and chooses a value V. Another proposer suggests a value and it gets accepted and so on. Essentially the value will not change and this is similar to electing the same leader over and over again. While anyone can start a proposal at any time, the agreed value will not get affected. However, the phase 3 or decide phase cannot happen if a new proposal with higher proposal number has started making rounds. Nodes may decide to reject the proposal and accept a new one. And this is possible since we can have multiple leaders. Therefore, there must be a gap between decide phase and new proposals for decide phase to happen. To re-iterate, this doesn't change the value however.

Q: What if you have same proposal numbers? Proposal numbers are unique, Paxos will not work if two proposals have same number. We can append PID (process id) to make it unique. This is similar to Lamport's clock ordering to convert partially ordered to fully-ordered events where we append process id.

Properties

- Property 1: any proposal number is unique.
- Property 2: two sets of acceptors have at least one node in common
- Property 3: value sent in phase 2 is value of the highest numbered proposal received in responses in phase 1.

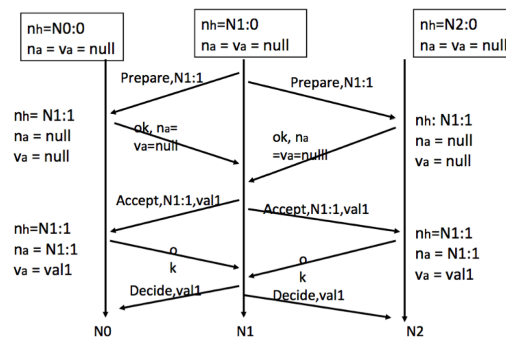


Figure 19.1: Example of Paxos with 3 servers

An example with three nodes namely N0, N1, N2 where N1 is the proposer:

- Prepare Phase:
  - N1 sends prepare messages to N0 and N2 i.e. <prepare, N1:1>where 1 is the proposal number and not the value we are trying to get consensus on.

- N0 and N2 haven't seen any proposals before so send <prepareok, n_a = null, v_a = null>to N1.

- Accept Phase:
  - Values received at N1 after prepare phase are null, so N1 decides on val1 as accepted value and sends accept messages to N0 and N2 as <accept, N1:1, val1>
  - N0 and N2 send <accept-ok>to N1.
  - Decide Phase:
    * N1 sends <decide, val1>to N0, N2.

When we have one leader, the protocol converges easily. But say N0 decides to become a leader while N1 is trying to get consensus as the proposer, the proposal from N0 will get discarded as a new proposal with higher proposal number is now available. Learners/Acceptors can choose to agree to the new proposal.

Issues :

- Network Partitions: For a network that has an odd-partition, if there is majority on one side, nodes can come to an agreement whereas they cannot if network is evenly partitioned.
- Timeout:
  - A node has max timeout for each message
  - Upon timeout, it declares itself as leader and restart Paxos
- Two Leaders:
  - Either a leader was not able to execute decide phase (due to lack of majority accept-oks as nodes encountered a higher proposal from other leader) OR,
  - One leader causes the other leader to use its value.
- Leader Failures: This case is same as two leaders or a timeout where a node will decide to become the leader and restart Paxos.

19.3 RAFT Consensus Protocol : understandable consensus protocol

The RAFT protocol is based on how a part-time parliament functions. A parliament is able to pass laws despite some members being out of attendance, or members showing up to the parliament at different times. It reaches consensus despite attendance (read failures, in case of processes).

Raft uses replicated logs or State Machine Replication (SMR) to implement the protocol. Assume we have n servers and each server stores a replica of log of commands and executes them in that order.

How do we replicate logs in multiple places while keeping the order consistent? Raft implements a leader election protocol. All incoming requests then go to the leader and it decides the order of execution and informs everyone, as opposed to sending each request to everyone and then deciding on an order. Therefore, we need to elect a responsible leader. And if leader fails, we elect a new one and clean the logs to ensure consistency. We must note that if we have majority i.e. N/2 +1 nodes, consensus can be reached, otherwise it cannot. Also, if an entry is committed, all entries preceding it are committed.

Log Replication Example: In case of three servers, the request z = 6 goes to the presumed leader. Leader writes it in log file and sends prompt to other nodes to append it to their logs. The consensus module ensures that the order is maintained. Every committed request is executed. The value needs to first be appended and then committed to the logs.

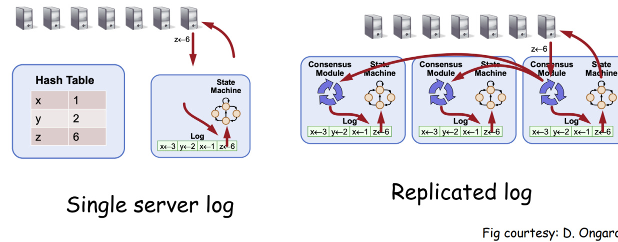


Figure 19.2: Example of Log replication

Consensus approaches:

- Leaderless/Symmetric: Client can send the request to any server and that server decides the order of execution.
- Leader-based/ Asymmetric: One server becomes leader and tells followers what to do.

Overview of RAFT operations

- Leader election: Nodes must select one server to serve as RAFT Leader. There must be provision to detect leader crash and provision to elect a new leader in case of a crash.
- Normal operation: This involves performing log replication, leader receiving client commands, appending incoming requests to log. Leader then replicates log to followers. We must ensure safety i.e., committed logs must not get impacted by leader crash and there must be at most one leader at a time.

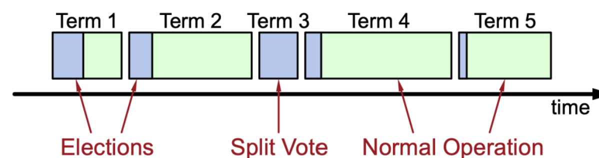


Figure 19.3: Terms

Terms:

- Time is divided into terms, a period when a certain node acts as the leader. Term does not change unless the leader crashes/fails.
- Each term has a blue followed by a green part. Blue parts represent leader election, green represents normal operation. If a term has only blue (a failed term), it represents a split vote or no majority to elect a node as the leader.

- All servers maintain the current term value.
- At any time, each server can be either of the three:
  - Leader: receives all client requests and does log replication
  - Follower: passively follows leader
  - Candidate: a node that participates in leader election

RAFT Election :

- Election timeout: Communication is over RPCs and if no RPCs are received for a while from the leader, then increment current term and become a candidate.
- Elections are selfish. On an election timeout, candidate node votes for self to become a leader and sends an election message (RequestVote RPC) to followers.
  - If the node receives vote from majority, it becomes the leader and sends heartbeat message (AppendEntries RPC) to inform other nodes.
  - Failed election: If no majority votes are received within election timeout, the term gets incremented and a new election starts.
- Safety in election: In any election, at most one server wins since you can only cast your vote once per term. Also, there is random back-off in case of a failed election i.e. each node backs off for different amount of time. This ensures that some node starts the leader election and wins majority, while other candidates are in timeout.
- Liveness: One of the nodes will win the leader election.

Normal RAFT Operation

- Leader receives client commands and appends them to log.
- Each log entry has 3 things: Index (item no. in the log), term (current term value), command.
- Leader sends AppendEntry RPC to all followers.
- Once an entry is safely committed to log (i.e. leader got a majority vote for AppendEntry RPCs sent), the command is then executed and results are sent to the client.
- Committed entries are notified to followers in subsequent RPCs therefore the followers catch up in background. The followers apply the committed commands to their state machines.

Log Consistency To verify if logs are consistent, leader informs the followers what the previous entry (index, term) in the log was. If the previous entry at the follower and the one sent by the leader do not match, then we know there is inconsistency. Log entries can become inconsistent due to leader failure.

- There can be missing entries as in the case of (a) and (b) followers. Possible causes can be a network partition or failure of those follower nodes when the entries came in.
- There can be extraneous entries as in the case of followers c, d, e and f. This can be because of leader partition, and some other nodes got new requests that haven't yet been committed.

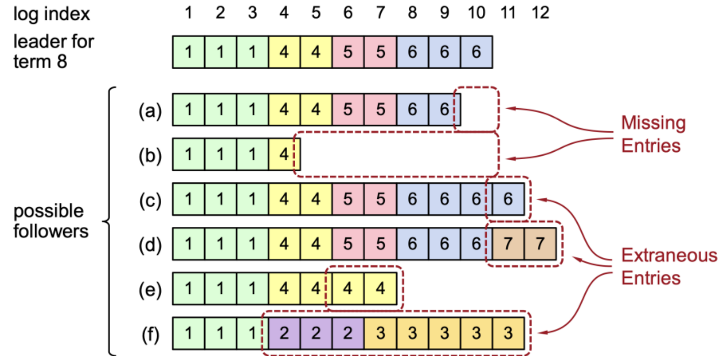


Figure 19.4: Inconsistencies in Logs Example

The leader must synchronize the logs to ensure consistency by adding required entries to the missing ones and scrubbing extraneous entries by using pre-fix match. **Note:** These are all entries that have been appended to logs but not committed.

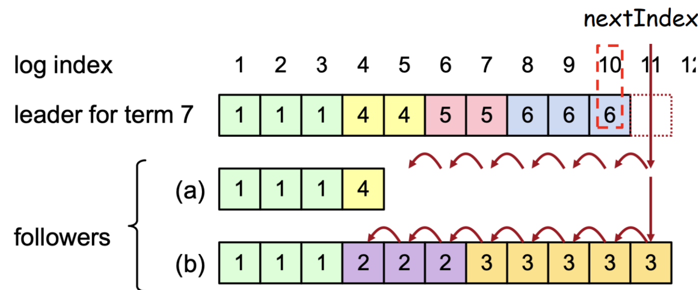


Figure 19.5: Log Repair Example

Log Repair The leader tracks nextIndex for each follower. It asks the follower if it has the entry at an index (index of last entry in leader's log) in its log. If the follower doesn't, the nextIndex decrements until a matching entry is found. All missing entries from this point onwards are sent to follower to catch up. In case of extraneous, the subsequent entries from index where we found the match at are deleted and leader replays the rest of the logs for follower to catch up on.

Leader crashed and some other node becomes the leader, how do we ensure consistency in this scenario? We check the committed entries until the time of crash and use that to ensure ordering.

When is consensus achieved in RAFT? Consensus is achieved when majority of the nodes have appended and committed the entries. To have consensus means we have agreed to commit to an order.

When does the commit actually happen? In Normal Operation, if majority of followers agree to append, then you commit the log.

During election, who ensures log is collected? The clients cannot send requests during election since there is no leader. Requests can be sent only after a leader is elected.

If the leader crashes while committing logs, what happens? RAFT has a way to handle this, TBA on piazza.

19.4 Recovery :

We have discussed techniques thus far that allow for failure handling, but how recovery dictates how those failed nodes come back up and recover to the correct state. The techniques include periodic checkpointing of states and roll-back to a previous checkpoint with a consistent state in case of a crash.

- Independent Checkpointing
  - Each process periodically checkpoints independently of other processes.
  - Upon failure, work backwards to locate a consistent cut, last checkpoint.
- Logging
  - Is a common approach to handle failures in databases, file-systems.
  - Done by logging and re-playing logs.

Trade-offs between checkpointing and logging: *Checkpointing* doesn't need logs, it saves system state that can be used as last consistent state. This is expensive since we are writing entire system state to disk. But recovery is quick in case of checkpointing, since we are loading the system values from a file essentially. Whereas in *logging*, the logs have to be replayed/executed again from the point of failure. Adding logs to a file is cheap, but it is expensive in terms of recovery as in the case of processes being behind by a lot and all the missed logs have to be executed again. We can combine the two as well.

- Take infrequent checkpoints
- Log all messages between checkpoints to local stable storage.
- To recover: replay messages from previous checkpoint. This avoids re-computations from previous checkpoint.

CMPSCI 677 Distributed Operating Systems **Spring 2022**

Lecture 20: April 11

Lecturer: Prashant Shenoy Scribe: ***Gayatri Belapurkar(Spring 2022), Mikayla Timm(Spring 2019)***

In this lecture, professor starts a new topic "Distributed Web Applications".

20.1 Traditional Web Based Systems

The web is basically a client server based global distributed system. The clients tend to be web browsers or application on the phone or other devices, which access the server component over HTTP. HTTP is a request response protocol, as also seen in Lab 2.

Figure 20.1 shows a basic client-server request response protocol, that exchanges static web pages. But the server side can consist of a complicated server that can process requests. In this image, the browser sends an HTTP request to the web server. The server fetches the document from database and sends back the response to the browser.

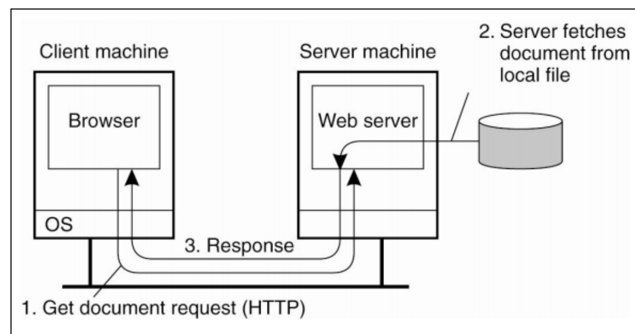


Figure 20.1: Overall Organization of a Traditional Web Site

20.2 Web Browser Client

Browsers are complex with many built-in functionalities. Web browsers have a user interface where the user can submit a request, the browser connects to the server, fetches a web page and renders that web page. As in the figure 20.2, there are multiple components such as user interface, browser engine (part that retrieves the content), rendering engine (take the content, decides the layout, etc.) and other components for network communication, client side scripts for interpretation (example: Javascript interpretation), and a parser that can parse HTML.

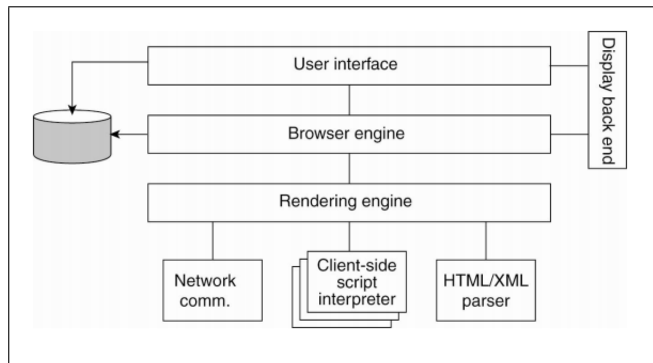


Figure 20.2: Logical Components of a Web browser

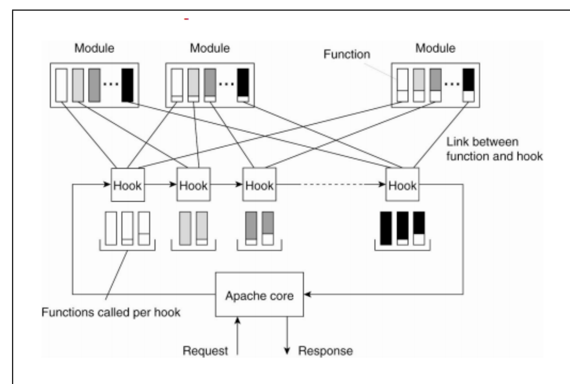


Figure 20.3: Overall Organization of Apache Web Server

20.3 Apache Web Server

The server also has complex components. Figure 20.3 is a Apache Web Server, one of the most popular web servers. The original version is a multiprocess model, other variants also support multithreading. By default, it uses the multiprocess model when starting up. The main process listens on HTTP and assigns incoming requests to child processes. Similarly, the multithreaded model can also be implemented with dynamic thread pools.

The architecture shows the process that takes place, irrespective of whether it is multithreaded or multiprocess. It has a modular architecture that uses pipeline processing. Processing is done by the various modules present (HTTP, SSL, etc.). These can be plugged in when the server is started. We need to configure a set of modules that will perform request processing. When a request comes in, it will go through a series of processing steps. Each module performs partial processing in a pipeline fashion. You can turn each module on or off. The only processing that is supported by the server is if we write the application using PHP. Other languages such as Java, Python will be processed outside the HTTP server.

20.4 Proxy Server

Proxy is an intermediary between the client and the server, making the client-server architecture a client-proxy-server architecture. Proxies can be used for multiple purposes. The example shown in figure 20.4 does protocol translation, converting from HTTP to FTP. Proxies are also used for web caching. A proxy is closer to the client and can process user requests faster than the actual server. Recently used web pages are cached. The browser will send requests to the proxy instead of server. Proxy will process the request. If the requested content is cached on proxy, it is will send back reply to client. Else, the proxy makes another request to the server.

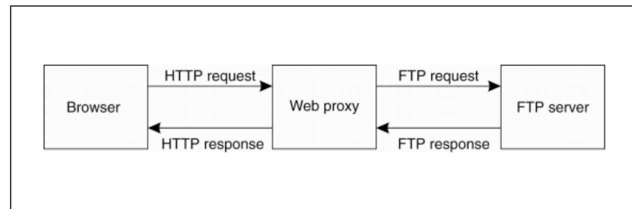


Figure 20.4: Web Proxy

20.5 Mutlitiered Architecture

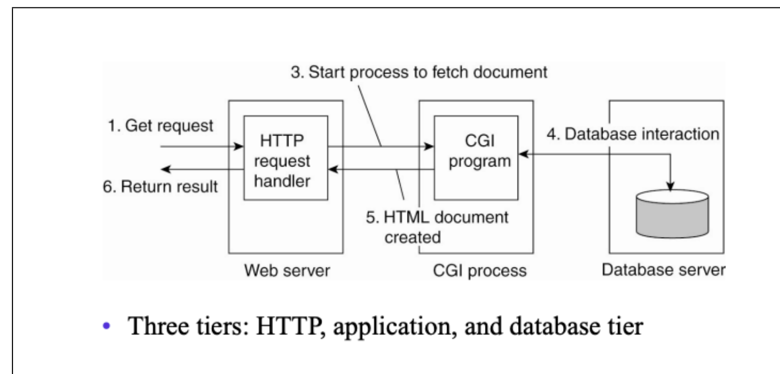


Figure 20.5: Multitiered architecture

The figure 20.5 represents a standard 3-tier web architecture wherein requests are sent from the web server, processed by the CGI program (which is now replaced by Python, Java, etc.) and then to the database. The request is processed and the response is sent back to the client. In this case as well, a proxy can be introduced to avoid repetitive computation.

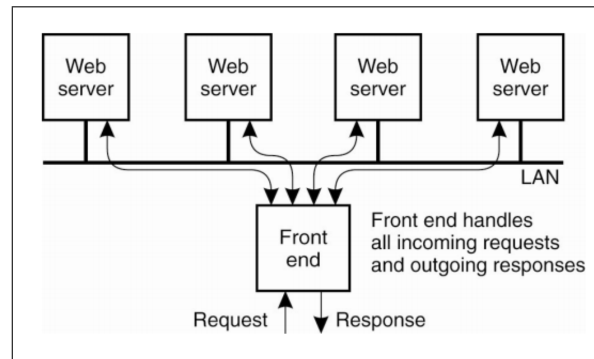


Figure 20.6: Web Server Clusters

20.6 Web Server Clusters

Figure 20.6 shows an example of clustered architecture. Each box in itself is multi-tiered. Each incoming request can be forwarded to one of the four Web Servers. The front end handles all the incoming requests and the outgoing responses. Responses flow back through the load balancing switch. This is a common way to scale applications. If a single machine is not enough, we replicate the web application. The clusters can service more requests, which allows it to scale up better.

There are 2 ways in which this can be implemented:

1. The load balancer receives all the HTTP requests and then forwards it to the app tier, whose multiple replicas are present. Each app tier replica has its own database replica. If the database is updated frequently, it also needs to be synchronized frequently. Thus, this is mostly used in databases that are read heavy and are infrequently updated. This supports distributed replicas.

2. The load balancer receives all the HTTP requests and then forwards it to the app tier, but all the multiple app replicas have a common database tier, that is they connect to the same component. This is more commonly used by multiple applications. In this case, no data synchronization is required. This works well as long as bottleneck is not I/O. In most cases, the bottleneck is request processing, thus this is not a problem. However, this way does not support having distributed replicas of the database, unlike method 1 above.

There are different ways in which the request can be forwarded to the servers in fig 20.6:

1. Request-based scheduling: Every HTTP request coming in can be potentially sent to any replica. This can be done in cases where there is no state to be saved. To facilitate the storing of a state in request-based scheduling, we can have a common storage where all the replicas can access the state of a client from.

2. Session-based scheduling: A browser first establishes a session with the web-server. Once this is done and maps to a replica, all requests of that server are sent to the same replica. This is beneficial as opposed to request-based scheduling as it helps in caching, helps in keeping the state in a single machine (example: saving the state of a shopping cart).

Irrespective of which way a request is forwarded, the way the load balancer can forward requests to one of the replicas, is either by using HTTP redirect, TCP splicing or TCP handoff. In the context of figure 20.7, the switch (load balancer) needs to send the request to some other server in order to process it.

1. HTTP redirect: The load balancer simply redirects the HTTP request to the web server, which then processes the request and sends the response back.

2. TCP Splicing: The client sends the HTTP request to the switch, which then makes another request to the web server. When the reponse comes back, the load balancer appends this to the first request and sends it back. These are essentially 2 connections that are spliced together.

3. TCP Handoff: Similar to HTTP redirect where we get a TCP connection and we hand it off to another machine. It requires network level changes to handoff using socket connections.

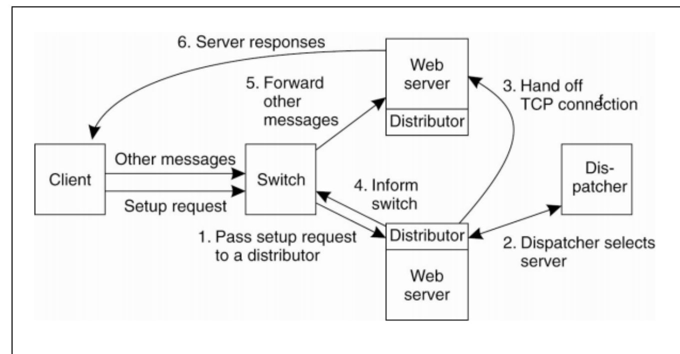


Figure 20.7: Scalable Content-Aware Cluster of Web servers

Question: Can you give some use-cases for these?
Answer: They all do the same thing. For HTTP redirect to work, all the machines would have to have a public IP address so that the client can connect to it. But if we have private IP addresses and only the switch has a public IP, we would not be able to use HTTP redirect, and would instead have to use splicing or TCP handoff.

Question: Will HTTP redirect have session information?
Answer: When any client sends a request to the switch, the switch keeps a table saying that it received a request from that client. It checks if this is a part of an existing session. If so, it takes that machine and does an HTTP redirect to it or sends the request using any of the other mechanisms. Thus, session/state and redirection mechanisms are somewhat orthogonal. However, to an extent, there are schedulers that would do an HTTP redirect and the client can send subsequent requests directly to that server.

Question: Suppose we receive an HTTP request and do some partial processing, and figure out that it needs to be sent to a subsequent process? Can we do the handoff then?
Answer: The multitier architecture does exactly that. We don't have to handoff the request, we can make another request and get its response. How to ask a client to forward all its subsequent requests to another server? This has other mechanisms, but even if some partial processing is done, we can still do an HTTP redirect request, as we haven't sent back a response yet.

Question: Is HTTP redirect a two-step process wherein we query the switch, it sends back an IP and then we send the request to this web server?
Answer: Yes, in HTTP redirect, we get a new URL to effectively talk to. In TCP splicing, we have the request already and make another connection and send it there, basically relaying the request. TCP handoff does something similar.

20.7 Elastic Scaling

It is an interesting technique that can be implemented when we have a clustered web application. Web workloads are time-varying (time-of-day effects, seasons when the workload is high, etc). There are other kinds such as load spikes or flash crowds, wherein the workloads increase suddenly (example: news story breaks). Some may be expected, such as sports events, big sales, and so on. How to deal with these changing workloads?

One approach is to decide the absolute maximum workload that the service will see and put enough servers to be able to handle it. But many-a-times, a lot of the servers would be sitting idle. Also, it is not always possible to predict the traffic. As a result, applications are generally under-provisioned, wherein the workload exceeds the capacity even when there are multiple replicas.

Elastic scaling/auto-scaling: Increase the capacity on the fly with increasing loads. The web server monitors its threshold and adds servers when this threshold is being approached. This can be done programmatically in cloud applications. There are 2 ways:

1. Horizontal scaling: There are multiple replicas and we add or remove those based on the load.
2. Vertical scaling: We don't change the size of the cluster, but change the size of the replicas, by giving them more cores.

This is used widely in modern cloud based applications. When do we scale? Look-ahead and predict workloads (maybe for the next hour) and pre-provision resources ahead of time. This is called proactive scaling. Another is reactive scaling. This is when we don't do any provisioning but monitor the load. For example, we only add new machines when the load reaches 70% or 80%. This may lead to small disruptions due to time taken to start the machines.

20.8 Microservices Architecture

Each application is a collection of smaller services. We take an application tier, which is basically a monolithic application tier and split it into smaller components, each of which is a microservice. This is also an example of a service-oriented architecture. This gives modularity and we can change each microservice independently, without affecting the other microservices and making it easier to manage. Teams can be responsible for one service. These can be independently deployed as well. Each microservice can be clustered and auto-scaled. Example: we can scale up only that microservice which is compute intensive. But this makes the application look more complicated. This is one more way of scaling web applications. The figure 20.8 shows 3 ways of scaling. The x-axis is horizontal scaling. z-axis is called data partitioning: we partition the data instead of replicating it. If we don't want to replicate our data but it becomes a bottleneck, one way is to split the data into parts and put it onto different machines. This is sharding or partitioning. The y-axis shows functional decompostion, wherein we take different microservices and scale them independently. We can use any combination of these, typically all the 3 are used.

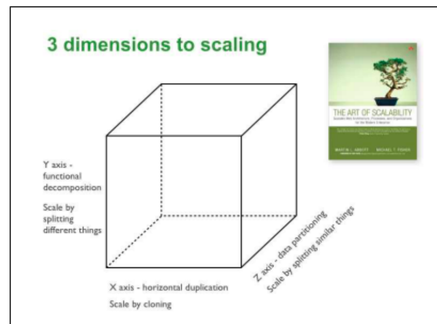


Figure 20.8: Scalability cube which shows 3 ways of scaling

20.9 Web Documents

Most web browsers get back content, mostly an HTML page with embedded content or objects. The figure 20.9 shows the types of objects. The simplest is the text object. The content is encoded using MIME. Web browsers extract it, parse it and render.

Web Documents

Type	Subtype	Description
Text	Plain	Unformatted text
	HTML	Text including HTML markup commands
	XML	Text including XML markup commands
Image	GIF	Still image in GIF format
	JPEG	Still image in JPEG format
Audio	Basic	Audio, 8-bit PCM sampled at 8000 Hz
	Tone	A specific audible tone
Video	MPEG	Movie in MPEG format
	Pointer	Representation of a pointer device for presentations
Application	Octet-stream	An uninterpreted byte sequence
	Postscript	A printable document in Postscript
	PDF	A printable document in PDF
Multipart	Mixed	Independent parts in the specified order
	Parallel	Parts must be viewed simultaneously

Figure 20.9: Web Documents

20.10 HTTP Connections

Figure 20.10 shows the original HTTP 1.0, where browser sets up a new TCP connection to the server every time we make a HTTP request. Each entity or object we get back has its own connection. Once we get a response, we tear down the connection. That is, it is non-persistent. Making new connection to the same server every time is wasteful. A variant, version 1.1 20.11 is more efficient as it creates persistent connections. We send next HTTP requests over the same connection. The browsers does not close the connection immediately, in anticipation of further requests. However, these are sequential. We need to get he response to the previous request before sending the new request. This can be slow and many browsers don't use this mechanism. For multiple simultaneous downloads, we set up multiple connections instead. Thus, HTTP1.1 did not solve its goal of saving connection overhead due to sequential requests.

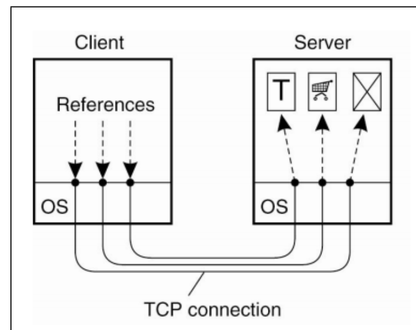


Figure 20.10: HTTP 1.0: Using Non-Persistent Connections.

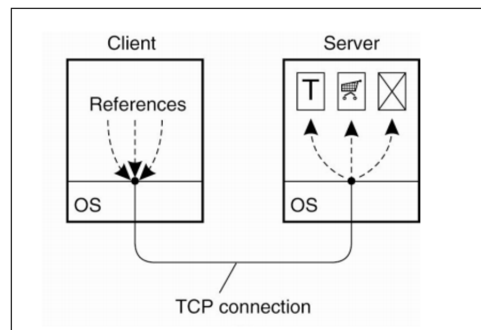


Figure 20.11: HTTP 1.1: Using Persistent Connections.

20.10.1 HTTP Methods

HTTP protocol has five methods:

1. The simplest among them is the 'GET' command. It takes a URL and simply fetches what the URL is pointing to, which can be an HTML page, an image, etc..
2. While 'PUT' command is used to store a document on the server.
3. 'POST' allows us to add data to the document. Whenever we submit webforms, it is a 'POST' request.
4. 'DELETE' can be used to delete a document, but most web browsers do not support it.
5. 'HEAD' gets the header of the document. It is typically used for caching.

HTTP 2.0 is designed to address the message latency problem. It allows us to have binary headers. We can compress headers and messages, making the message smaller and thus faster. It allows concurrent connections (persistent but with concurrency), thus we do not have to wait for responses for the previous requests. This is done using the concept of streams. Each stream is one request and one response. To send a request, we send it using a new stream, thus when the response comes, we know which request it is intended for. This helps in speeding up the connection significantly. Both the browser and the server have to support HTTP 2.0. It is not backward compatible.

20.11 Web Services Fundamentals

Webservices are ways in which we can write applications and use RPCs between these applications or between a client and a server. The term webservice has a specific connotation where we use a certain type of interface description language, a certain protocol for SOAP for us to send RPCs. We use an interface definition language called webservice definition language (WSDL), a compiler that generates stubs for the client and the server, the protocol SOAP (like HTML but is XML based) used for communication. This can be seen in fig 20.12.

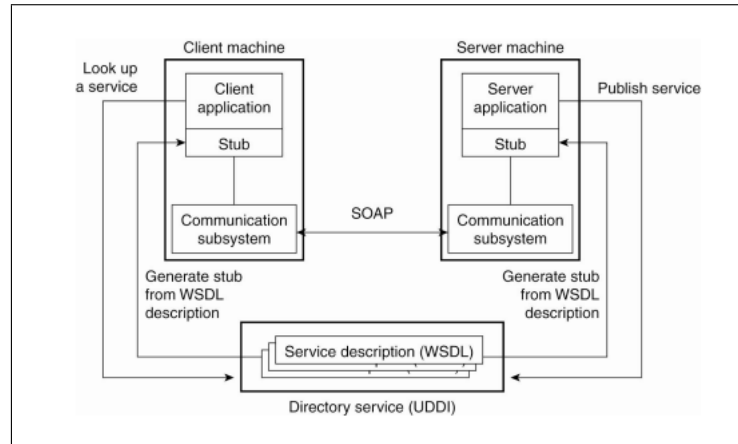


Figure 20.12: Web service

SOAP, Simple Object Access Protocol, was used to make RPC requests over HTTP. Fig 20.13 is an example of making an RPC request over SOAP. The entire RPC request is sent as a XML document. The server after receiving the document, parses it, perform required operation and sends back the response as another XML document. Figure 20.13 shows one such XML request document. Here the client is calling 'alert' method and passing the string 'Pick up Mary at school at 2pm' as an argument to that method.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
     <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
       <n:priority>1</n:priority>
       <n:expires>2001-06-22T14:00:00-05:00</n:expires>
     </n:alertcontrol>
   </env:Header>
   <env:Body>
     <m:alert xmlns:m="http://example.org/alert">
       <m:msg>Pick up Mary at school at 2pm</m:msg>
     </m:alert>
   </env:Body>
</env:Envelope>
```

Figure 20.13: Example of XML-based SOAP message

Question: While communicating between microservices, do we use HTTP 2.0 or HTTP 1.0? **Answer:** To some degree, it depends on what the HTTP library does. The code should not change for this. It should be HTTP 1.0 as HTTP 2.0 is mostly used in browsers.

Question: If we use HTTP 2.0 library and make a new connection, it should not create a new one, right? **Answer:** If we use HTTP 2.0 library and make a new connection, internally it should check that there is already a connection and should not create a new one. Another way is we don't call it repeatedly to create, but just send the request over the existing socket.

20.12 Restful Web Services

As we can see, for calling single method with one argument, we have to send such a long XML file. SOAP did not perform well because of this overhead. As a result, SOAP evolved into Restful architecture. Restful architecture makes RPC request over HTTP. HTTP was already popular than something like SOAP, so it was chosen as a way to make RPC requests. In case of Restful architectures, the communication is light weight and assumes one-to-one communication between client and server.

In Resfult webservices, we use:

1. GET: to read something
2. POST: to create, update or delete something
3. PUT: to create or update something
4. DELETE: to delete something

The example in 20.14, a GET request is made and the response is received. The response sends an XML packet in this case, but could also be JSON, any format can be used. It is much more compact than using SOAP.

```
GET /StockPrice/IBM HTTP/1.1
Host: example.org
Accept: text/xml
Accept-Charset: utf-8
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<s:Quote xmlns:s="http://example.org/stock-service">
     <s:TickerSymbol>IBM</s:TickerSymbol>
     <s:StockPrice>45.25</s:StockPrice>
</s:Quote>
```

Figure 20.14: Example of a Restful web service

20.13 SOAP VS RESTful WS

1. SOAP application can be written in various languages and can run on different OS or platforms and is also transport agnostic (XML, TCP, etc. cab be used). It need not use HTTP. Whereas, RESTful services only support HTTP. Restful services, however, can use any language to write this, such as Python, Java, and so on.

2. SOAP is for general purpose distributed systems. We can have all kinds of applications making calls to each other. Restful webservices have a client and a server and is point to point. We have pairwise interactions, just like traditional RPC.

3. SOAP has a wide set of standards (telling us how to write, compile, etc/), whereas RESTful services does not have any pre-defined standards. They have general guidelines, but no one particular way to write it.

4. SOAP is very heavyweight compared to REST.

5. Rest has less of a learning curve compared to SOAP.

20.14 Web Proxy Caching

One mechanism to use proxies is to use for web caching. Typically, we take a URL and send it to the server that answers to that URL. Let's say there is a proxy server sitting near the client, which maintains a cache. All requests are sent to the proxy server, which looks at the cache. If data is found, we immediately send back the response. This helps in faster responses for the client if the proxy is near the client and reduced loads for the server. If we cached a webpage and the webpage changed on the server, the proxy may serve stale content. Thus, cache consistency is important.

Suppose if we are looking for some webpage but it is not there on the proxy, this is called a cache miss. There are 2 things to do. One is to and get it from the server, put it in the cache and send the response. Figure 20.15 shows another method to deal with this, called "Cooperative Caching". Along with communicating with the server, these proxy caches can also communicate with each other. A proxy reaches out to the near by proxies to get the data. If the data is present on nearby proxies, it is faster than reaching out to the server. The client sees the union of all the stored caches.

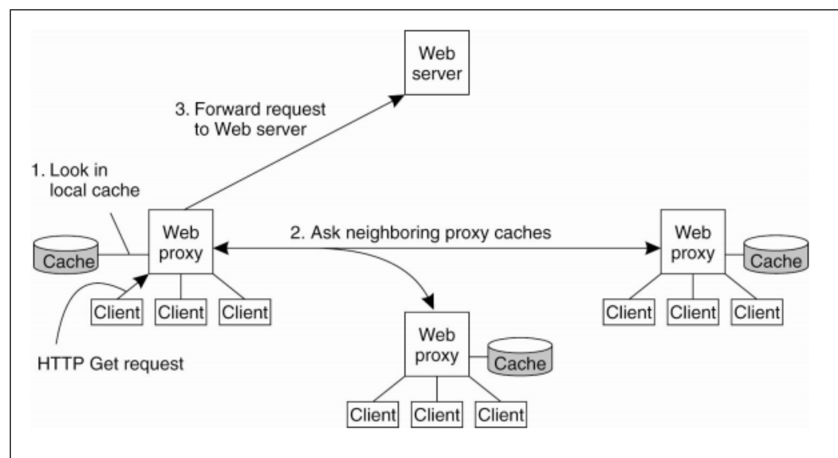


Figure 20.15: Cooperative Caching

20.15 Web Caching

It is also import to deal with consistency. Web pages tend to change with time. When a browser fetches a page from the server, we are guaranteed that the returned page is the most recent version. While using proxies, we need to ensure the consistency of cache web pages. The popularity and update frequency can be different across web pages. We need to consider both these issues for maintaining consistency. That is, if it is a cache hit, how will the proxy know it is the updated data?

1. Pull based approach: We poll periodically and use a conditional GET to ask the server if the cached data has changed

2. Push based approach The web server tracks each proxy and the pages cached there. If there has been an update, it sends a push to invalidate the data.

CMPSCI 677 Distributed & Operating Systems Spring 2022

Lecture 21: April 13

Lecturer: Prashant Shenoy *Scribe:* ***Mikayla Timm (2019), Saloni Chalkapurkar(2022)***

21.1 Web Caching Continued from the previous lecture

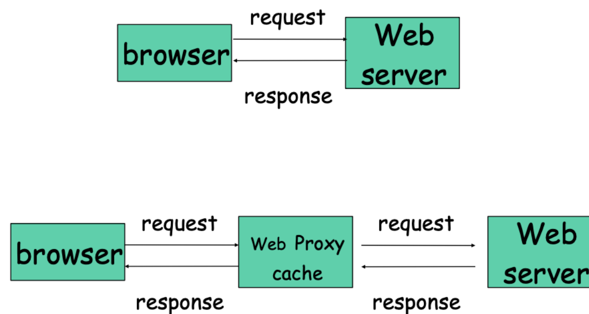


Figure 21.1: Client-Server and Client-Proxy-Server Architectures

Web caching uses client-proxy-server architecture. Clients send requests to the proxy. Proxies can service the requests directly if they have the resources. If they do not have the resources, they go to the server to get the request processed. In web caching, the proxy provides the service of caching i.e. the proxy caches content from the server and when the client browsers make a request, if the content is already in the cache, the content is returned.

21.2 Web Proxy Caching

The discussion for this section assumes a collection of proxy caches sitting in between the client and the server. Along with communicating with the server, these proxy caches can also communicate with each other. This mechanism is called "Cooperative Caching". Figure 21.2 shows one such scenario where a client sends a request to the web proxy. The web proxy will then look into it's local cache for the requested web page. If it is a hit, then it will send the response back. In the case of miss, typically the web proxy will contact server. But in the case of cooperative caching, cache misses can be serviced by asking nearby/local proxy instead of the server i.e. it will reach out to the near by proxies to get the data. This can make fetching faster than getting data from server. Caches are cooperating with each other and clients see union of all the content stored in nearby caches rather than just the content cached in the local proxy.

21.3 Consistency Issues

What can be done if content in the proxy becomes stale? When content is updated at the server, the proxy can no longer serve the content to the users because they will see an outdated version of the webpage.

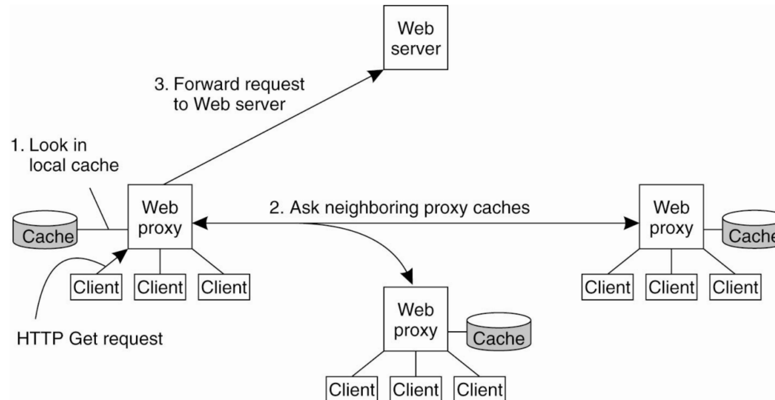


Figure 21.2: Web Proxy Caching

Web pages tend to change with time. Update frequency of web content varies from page to page. When a browser fetches a page from the server, we are guaranteed that the returned page is the most recent version. While using proxies, we need to ensure the consistency of cache web pages. How can the proxy maintain consistency of web content when different webpages get updated at different frequencies? There are 2 approaches for this - pull based and push based. Regardless of the approach, the proxy is notified either by sending an invalidate message or by sending the new updatedversion of the page.

21.3.1 Push based Approach

This approach relies on the server. It is the responsibility of the server to ensure that all the content stored at various proxy caches are never out of date. For every webpage stored at the server, the server keeps a table that lists all proxies that have a cached copy of that webpage. If a page is updated, server looks at the table and finds all the proxies that have a copy of the page and notifies the proxies that the page is updated. This notification can be of 2 types -

1. **invalidate** - simply inform the proxy that the webpage has changed and discard it from the cache.

2. **update** - send the new version of the page which automatically tells the proxy that the page was changed and the new page can be put in the cache and delete the old version.

When to use invalidate and when to use the update message?
Invalidate is a short message which simply notifies whereas update is large message which send the entire content of the updated page. If the content is popular at proxy, it makes more sense to update as it will take time when proxy has to fetch the page from the server again when a subsequent request for it is made. If the content is not popular, it makes more sense to send the invalidate message as update might waste bandwidth by sending the page to the proxy but the proxy has no request for the webpage subsequently.

Push-based approach provides tight consistency guarantees. Proxies can also be passive, as the server does all the work in this approach. Server becomes stateful - for every webpage it keeps track of proxies. One disadvantage is that, since HTTP is stateless, you need mechanisms beyond HTTP. Another disadvantage is that the server needs to keep track of proxies forever.

21.3.2 Pull based Approach

Relies on the proxy. Proxy is responsible to maintain consistency. It polls the server periodically to check if the previously cached page has changed. Polling is performed using conditional GET (if-modified-since HTTP messages). That is if the web page has changed since the timestamp 't', GET me the updated web page. Will return modified webpage only if it has changed.

When to poll for webpages?
Depends on the frequency of updation. For web pages which change very rarely, frequent polling is extremely wasteful. There are no such wasteful messages in the push based approach. If the page changes much more frequently than the polling frequency, then the proxy might cache outdated content for longer times. Thus, polling frequency should be decided based on the update frequency of web page. There are two ways to do this:

1. Server can assign an expiration time (TTL: time-to-live values). This time is an estimation of next possible changes on the web page. Server can estimate it based on the past web page update frequency history. It is likely that webpage might change after TTL so poll after TTL expires.

2. Proxy has intelligence to dynamically figure out the polling times. Poll duration is varied based on the observed web page updates. Dynamically change polling frequency to understand average rate at which webpage is changing.

Question: Can this change be done like using congestion control techniques like AIMD(Additive Increase Multiplicative Decrease) to figure out congestion window using network congestion?
Answer: This idea was directly inspired by congestion control techniques where you are probing in that case the network to figure out the congestion window, here you are probing the server.

Question: Do we need to worry about clock synchronization?
Answer: Not too much, assume clocks use NTP accuracy of 10 miliseconds. As webpages wont change every 10 miliseconds (usually changed in days or weeks), need not worry about synchronization.

Advantages and Disadvantages
Pull based approach gives weaker consistency guarantees. Updates of a web page at server, might not be immediately reflected at proxies. Latency to sync the content might overtake the benefits of proxies. There is a higher overhead than server push approach, and there could be more pulls than updates. Some advantages are that the pull based approach can be implemented using HTTP (server remains stateless). This approach is also resilient to both server and proxy failures.

Question: Pull based approach does not have persistent HTTP connections is that a limitation of this approach and of server push has persistent connection?
Answer: Neither approach requires persistent connections. If polling frequency is less does not make sense to keep persistent connections - wasting lot of resources. Only makes sense to do this when content is changing frequently.

21.3.3 A Hybrid Approach: Leases

Hybrid approach based on both push and pull. 21.3 shows its worrking.

Lease is a contract between two entities(here server and proxy) in the distributed system. It specifies the duration of time the server agrees to notify the proxy about any updates on the web page. Updates are no

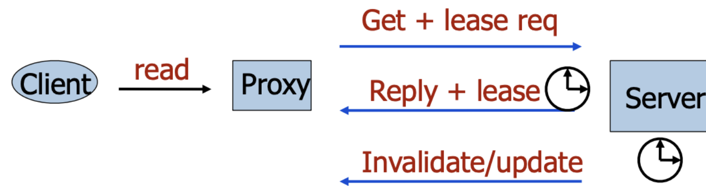


Figure 21.3: Lease

longer sent by the server after lease expiration and the server will delete the state. Proxy can renew the lease. If the page is unpopular, proxy can decide not to renew the lease for that page.

Lease is a more limited form of server push - performing push for the duration of the lease only. We get advantages of push and do not have to keep state indefinitely. If the lease duration is zero, degenerates to polling (pull), if duration is infinite it is server push and makes the server stateful. Duration in between zero and infinite is a combination of both pull and push.

Tight consistency guarantees when lease is active.

21.3.4 Policies for Leases Duration

Lease duration is an important parameter. There are three policies for lease duration:

1. **Age-based:** Based on frequency of changes on the object. The age is the time since last update. Assign longer leases for more frequently updated pages.

2. **Renewal-frequency based:** Based on frequency of access requests from clients. Popular objects get longer leases.

3. **Server load based:** If the load on the server is high, we should use shorter lease duration. This will remove the burden of storing proxy state on server side.

Question: Does each proxy have its own lease?
Answer: Not only does each proxy have a lease, there is a lease for each webpage at a proxy. If a proxy caches 100 different webpages, each of them will have a different lease. It is not lease per proxy, it is lease per webpage at a proxy. Different proxies will have different leases for the same page.

Question: Is it the proxy's responsibility to keep track of the lease or the server's responsibility?
Answer: Both. Server has to keep track of all active leases and send notifications whenever the webpage changes for every active lease on that webpage. Proxy has to track lease as well. If the lease expires,proxy decides whether to renew it or not.

Question: Do you need a separate monitoring framework to keep track of popularity of content?
Answer: Server by itself will not know how popular the content is. Sever can know the age and server load. Proxies can track popularity of the webpage and report stats to the server. Monitoring framework can be added to the proxy system, it is not a part of lease approach.

21.3.5 Cooperative Caching

Collection of proxies cooperate with each other to service client requests. We will assume that there is a heirarchy as tree structure with leaf caches. Figure 21.4 explains the message flow for once single client request in the case of "Hierarchical Proxy Caching". The client requests for a web page. If it is a cache miss, the proxy will send requests to it's peers(red arrows) and parent using ICP(Internet Cache Protocol) messages. If none of the peers have it, they will send back the non-availability response to the proxy(green arrows). The proxy will then send an HTTP request to the parent to resolve. If there are more levels in the heirarchy, the process will repeat for its parents and so on. The parent will fetch the data from the server and sends the web page as response - flows down the heirarchy to the client.

This works well when a nearby proxy actually has the content. If there is a global miss - no one in the heirarchy has the content, latency will increase significantly. As we can see, there is a lot of messaging overhead. Also, browser has to wait for longer times in the case of cache miss on the proxy. This will affect performance. Latency could increase - it may have been faster to just directly request from server in the event of a cache miss on the proxy.

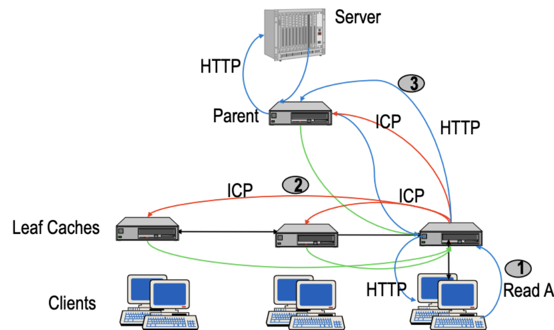


Figure 21.4: Heirarchical Proxy Caching

This is complicated as seen in the figure, as there are many arrows all over the place. To simplify, we can flatten the hierarchy. Every proxy keeps track of what is stored in its nearby caches. Keep track of what's cached using table lookup - save which node has what in the table. If the content for a request in a nearby cache, it is fetched. If there is no entry in the table for a request i.e. none of the nearby caches have the content, go to the server to get it, and add it to the table. Now there is no heirarchy, it is a flat network of proxies. Here, lookup is local. Hit is at most 2 hops and miss is also at most 2 hops. Every time you fetch or delete a page, you update the table for all the nodes. Every node keeps a global table that must be kept consistent. There is an additional overhead for keeping this consistent but performance is better.

Question: In the server push approach or lease approach, can you use multicast to notify all proxies?
Answer: The notifications can be sent as either n unicast messages one to each proxy that has the content or a single multicast message where all proxies are listening so that is a more efficient way of sending messages. That is orthogonal to whether lease is being used or not
Follow-up Question: If you do multicast, you don't require lease?
Answer: The reason a lease is required is if you don't have a list of which proxies to notify and send an update to every proxy in the system, maybe only 10 out of 10000 proxies have the content. You have now wasted messages by sending message to 10000 proxies. Even though it is a multicast message, you are still using network resources to send it. If you want to multicast, you want to send it only to the proxy group that has the content and so you need to track that.

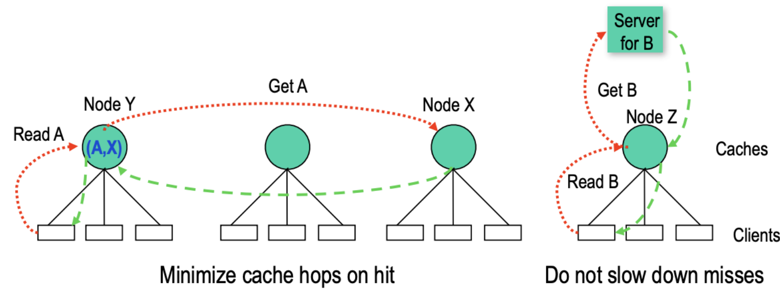


Figure 21.5: Locating and Accessing Data in the Flattened Network

Question: If you want to do multicast, how to identify proxies that have the content?
Answer: You would have to construct a multicast group for every webpage. When a proxy caches that content, put that proxy in that group. When the content is removed from proxy, remove proxy from the group.

Question: If you have heirarchical proxy caching system, can different proxies use different consistency mechanisms - some push some pull?
Answer: That would be a problem. Since caches are interacting with each other, it is better to use uniform consistency mechanism.

21.4 Edge Computing

It is the evolution of proxy servers into a more general approach where servers are deployed at the edge of the network and they provide a service. These servers can provide more than just caching services. Applications can be run on these servers. Edge computing is a paradigm where applications run on servers located at the edge of the network. Benefits include lower network latency than remote cloud servers, higher bandwidth and can tolerate network or cloud failures.

Cloud computing platforms are treating edge computing as an extension of cloud computing where cloud resources are being deployed closer to users rather than in far off data centers.

21.4.1 Edge Computing Origins

Edge computing evolved simultaneously from mobile computing and web caching.

Content Delivery Networks - As web caching became popular, several commercial providers offered proxy caching as a service - they deployed proxy caches in many different networks. If you were an operator of a web app you could become a customer and they could cache your content and deliver to your customers for a small fee. These companies deployed CDNs - large network of proxy caches deployed all over the world. You could offload your content to these proxy caches and deliver to end users at low latency. This network of caches was an early form edge computing.

Mobile Computing - Early mobile devices were resource and energy constrained. Not advisable to do heavy computations on these devices. One approach was to put servers near the edge of the wireless network. The work was offloaded to the edge server. This was called computation offloading - offloading work from one

device(mobile device) to another(edge server). This approach was also an early form of edge computing by offloading computation at low latency.

21.4.2 Content Delivery Networks (CDNs)

Global network of edge proxies that provide caching services among other services to deliver web content. Useful to deliver rich content like images or video content which can increase the load on server significantly as its better to cache such content to reduce load on server. Commercial CDNS deploy lots of these servers in many different networks. Servers are deployed as clusters of different sizes depending on the demand.

Content providers are customers of CDN service. They decide what content to cache. Consistency is the responsibility of the content provider.

Users access website normally, the content is fetched by browser from CDN cache.

21.4.2.1 CDN Request Routing

Because they have lots of different servers, when a request comes in, CDN has to decide which proxy has to serve that request. Request for the same content is sent to different proxies based on location of the user. The idea is to send the request to the server nearest to user. Request routing is routing the request to the right proxy to get it served at the lowest latency. How do CDNs do this? They have large load balancers that look at incoming requests and send it to different proxies. Typical CDNs have 2 level load balancer:

1. gloabl level - decide whcich cluster to send the request to. nearest cluster is chosen
2. local level - once request is mapped to a cluster, which server in the cluster will serve the request.

Deciding which is the closest cluster or closest server is done using Domain Name Service (DNS). DNS is a service which takes the url and gives the IP address. The browser makes a connection to this returned IP address. DNSs used by CDNs have special algorithms that change the IP address returned to the client based on the location of the client.

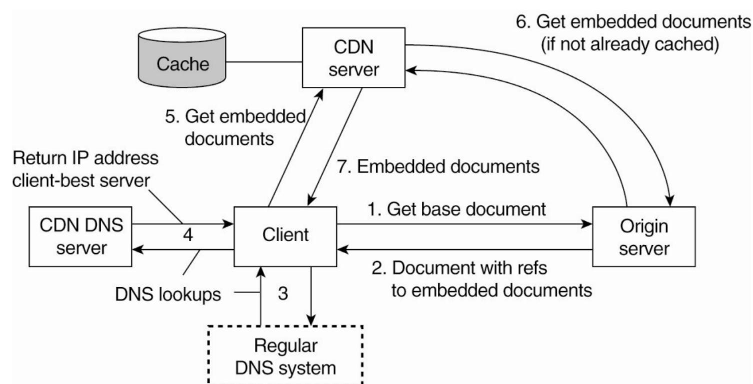


Figure 21.6: CDN Request Processing

Question: Do edge proxies use cooperative caching?
Answer: In this case, no need to do cooperative caching. Essentially going to replicate content and local load balancing will ensure that the server you are getting mapped to has a copy of the content.

Question: Are DNS and CDN independent services?
Answer: In this case, the DNS is run by the CDN server. Your browser will send a request to your local DNS. That DNS server will send that request to another server responsible for the domain you are going after. For example cnn.com/newsvideo.mp4. Cnn would in this case have the CDN run its DNS service for it so then the request goes to CDN where all of this happens i.e deciding the closest server. DNS is indeed separate from CDNs but some DNS servers in this case are going to be run by CDN and those are the servers for domains that it is actually caching content for.

Question: Is there criteria apart from geographical proximity that is used to do load balancing?
Answer: This is the case. Proximity is not the only criterion, there will be a lot more sophistication to handle overload etc. For example - fault tolerance has to be built in.

CDNs have evolved from simple caches to running entire applications at the edge. Figure 21.7 shows CDN hosting web apps. Dynamic content is not cacheable so caches are less useful for CDNs. So CDNs allow running applications on edge server at low latency.

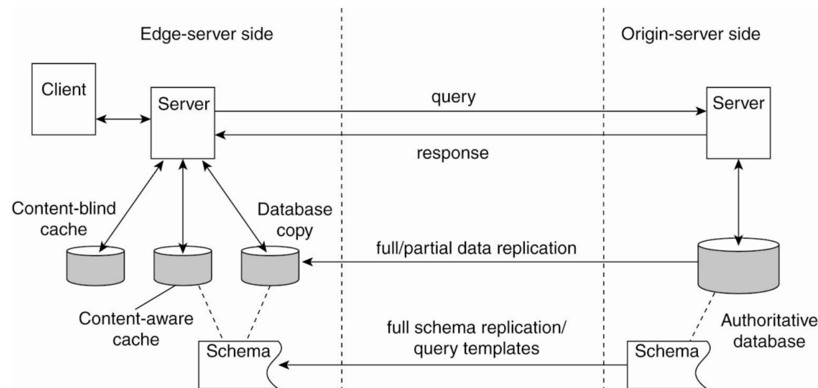


Figure 21.7: CDN Hosting web apps

21.4.3 Mobile Edge Computing

Allows mobile devices to offload compute-intensive tasks to edge servers. Use cases are mobile AR/VR where the mobile device had to process graphics heavy content that drained battery life faster and heavy duty computation was required. Since users are interacting with the system, low latency is very important. Edge servers provided both compute power as well as low latency.

Mobile devices today are much more capable and need to offload from smartphones has reduced. Other devices today that are not as capable such as headsets still use mobile edge computing for offloading compute intensive tasks.

CMPSCI 677 Distributed & Operating Systems **Spring 2022**

Lecture 22: April 20

Lecturer: Prashant Shenoy *Scribe:*

22.1 File System Basics

22.1.1 File

A file is a container of data in text format, binary format etc. which is stored on a disk so that the user can re-visit it at a later point in time. In UNIX, a file is an uninterpreted sequence of bytes which implies that the file system is unaware of the contents/type of the file. Other operating systems like Windows and Mac knows the file types (This information can be useful to open a file in the right application).

22.1.2 File System

- File system abstracts and provides a logical view of data (a hierarchy of files and folders) and storage functions.
- It helps us to create, modify, organize and delete files and takes care of how to map them to the underlying storage device.
- It provides a user-friendly interface so that the user need not deal with the low-level interfaces exported by the disk.
- It allows us to share the files among other users by giving permissions and also allows us to protect the files.

22.1.3 UNIX File System Review

- In UNIX, the files structure can be viewed as a directed acyclic graph. Note that this looks like a tree structure but can contain soft links pointing from one directory to other which makes it a DAG. Each directory entry for each file contains the file name, inode number (metadata for the file), major device number and minor device number. All inodes are stored at a particular location on the disk called super block. To access the file, the file system needs to first get this metadata to know where the file is located in the actual disk (aka block locations of the file).
- An inode structure consists of the fields like mode, Owner ID, group id, Dir file, protection bits, last access time, size, reference count, address[0]...address[14] etc. The addresses stores the pointers to the data blocks. The first 12 are the direct blocks which stores the pointers to the data blocks (see figure 22.1), the 13th address stores the pointers to the location which in turn stores the pointers to the direct data blocks (one level of indirection). The 14th address follows two levels of indirection which stores the pointers to one level indirection blocks. So the hierarchy grows as the size of the file grows but we have an upper limit of the size of the file that can be stored on this file system because we only have a certain number of pointers in addresses (In this case from 0 till 14).

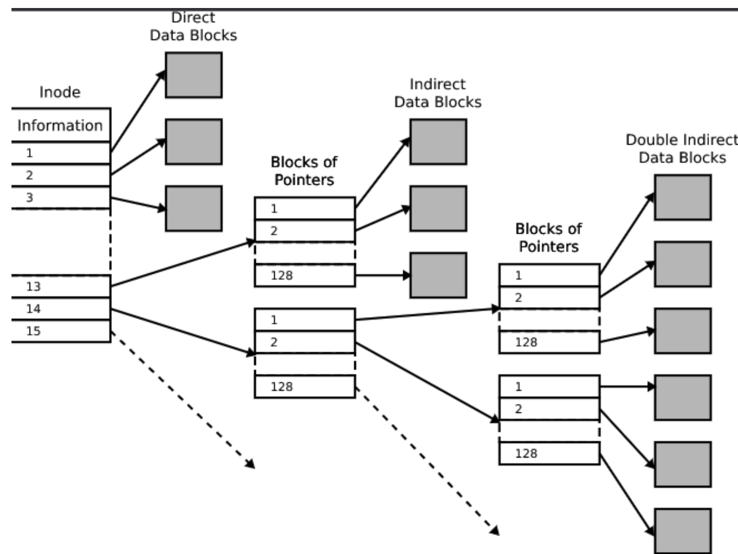


Figure 22.1: Inode structure

22.2 Distributed File Systems (DFS)

If files on a different machine can be accessed, it is a distributed file system. Another way to think about DFS is that the servers store different files on different servers, and all the servers collectively form your file system.

22.2.1 File server

A machine that stores all the files.

22.2.2 File service

The interface that the machine exposes for other machines to access the files on this machine. For example NFS uses RPCs to send read/write requests to a remote file system. There are two types of file services as shown in figure 22.2.

- Remote access model: The client requests are sent to the server and the server sends back the results after doing the work requested by the client.
- Upload/download model: When the client performs a request to the server, entire file is sent as a copy to the client, and subsequent access are made to the local copy. To maintain consistency, the client eventually sends back the changed file to the server.

Note: As the files are directly updated on the server, there is consistency in the remote access model, but each operation is an RPC call which makes it slow. In upload/download model, there is a period of time in which the file on the server is out of date. Having said that, upload/download model gives better performance as the operations are taking place on the local machine and very less calls are made to the remote machine.

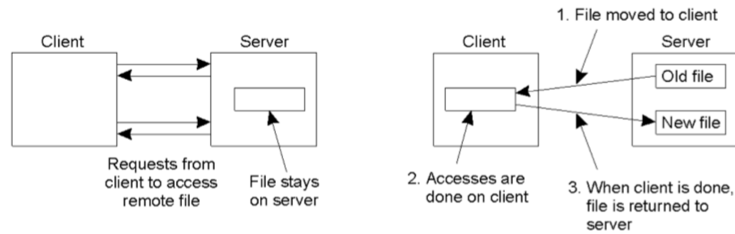


Figure 22.2: Remote access model(left), Upload/download model(right).

22.2.3 Server Type

There are two types of server and one would need to make the choice from one of them when building any distributed file system.

Stateless server: No information about clients is kept at the server.

Stateful server: Server maintains information about the client accesses. It is less tolerant to failures because the state is lost when a server crashes. There is slight performance benefit here due to the compact request messages (Clients do not need to send the information like permissions every time the request is being made). Consistency and idempotency are easier to achieve.

Note: An Idempotent server executes as if it has performed the request only once regardless of how many times same request was received.

22.2.4 Network File System (NFS)

NFS is a layer on top of an existing file system that allows to share the file system over a network. NFS is implemented using virtual file system layer supported by the underlying operating system. Virtual file system layer can be seen as a forwarding layer that looks at where is the file stored and invoke that file system(local file system for local files and possibly NFS for remote files).

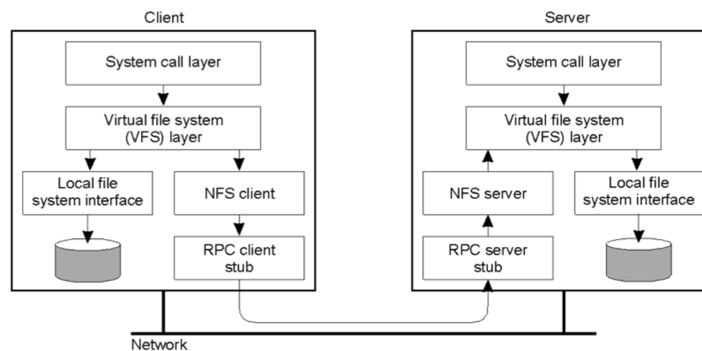


Figure 22.3: Network File System (NFS)

Note: Till version 3, NFS used stateless server protocol but from version 4, it uses stateful server protocol. So it now supports open call to a remote file.

In figure 22.4, we can observe that in version 3 of NFS, individual LOOKUP and READ RPC calls were

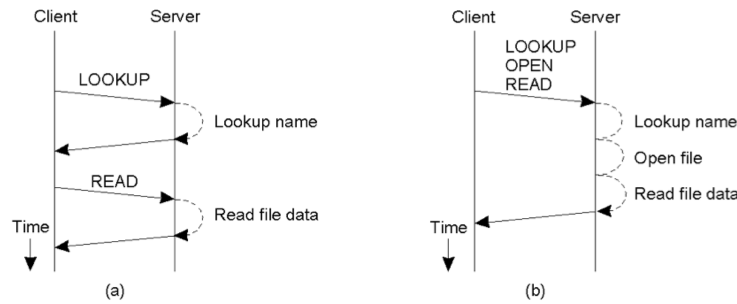


Figure 22.4: Difference in communication in NFS version 3 and version4.

needed whereas in version 4 of NFS, we can perform a batch RPC request.

22.2.5 Mount protocol

Mount protocol is a way how a NFS client gets access to a remote file system. Certain directories can be mapped from remote file system to the local file system in order to get access.

22.2.6 Crossing mount points

Crossing mount points is mounting nested directories from multiple servers.

22.2.7 Automounting

Automounting is also known as mounting on demand. The mappings get established but the mounting only happens when the user tries to access those directories. And if there is an idle time, it unmounts. This way we can reduce the amount of kernel resources used.

22.2.8 File attributes

There are specific attributes (like TYPE, SIZE, CHANGE, FSID) that a file system must to support to be compatible with NFS. There are other attributes (like OWNER of a file) which are not mandatory to be compatible with NFS but are recommended.

22.2.9 Semantics of file sharing

- In UNIX semantics, every operation on a file is instantly visible to all the other processes using the same file.
- In session semantics, no changes are visible to other processes until the file is closed.
- Immutable files cannot be mutated. A new version of the file needs to be created if we need any changes.
- In transaction semantics, all changes occur atomically.

Note: NFS follows semantics in between UNIX and session. It caches the file and periodically flushes the changes to the server. If one process writes to a file, the other process might have a different or outdated version of the file for a period of time. NFS uses local caches for performance reasons which leads to this weak consistency.

22.2.10 File locking in NFS

- Version 3 of NFS used stateless server protocol. One of the uses of having a state is file locking. Version 4 of NFS uses stateful server protocol, so applications can now use locks to ensure consistency. File locking can be done in different ways like locking the entire file, locking a specific range of bytes in a file etc.
- In share reservations file locking, we have the notion of a denial state where if an application has a write denial state to a file, it cannot write to the file but it can read the file.

CMPSCI 677 Distributed Operating Systems **Spring 2022**

Lecture 23: April 25

Lecturer: Prashant Shenoy *Scribe:* ***Jeff Mao***

In this lecture, continue the topic ”Distributed File Systems”, Case Study: Coda File System and xFS

23.1 Coda Overview

23.1.1 DFS designed for mobile clients

- Nice model for mobile clients who are often disconnected

- Use file cache to make disconnection transparent
- At home, on the road, away from network connection

23.1.2 Code supplements file cache with user preferences

- E.g., always keep this file in the cache
- Supplement with system learning user behavior

23.1.3 How to keep cached copies on disjoint hosts consistent?

- In mobile environment, ”simultaneous” writes can be separated by hours/days/weeks

Question: What is coda using a remote access model or an upload download model?

Answer: It’s a little bit of both. When you’re connected, your changes can be uploaded or sent to the server immediately, but you always have a cash. So when you are disconnected, you’re essentially just working with whatever files subcaste, in which case you it looks like an upload download model. So the answer is it actually depends on whether you’re the state of.

23.1.4 File Identifiers

- Each file in Coda belongs to exactly one volume
  - Volume may be replicated across several servers
  - Multiple logical(replicated) volumes map to the same physical volume
  - 96 bit file identifier = 32 bit RVID + 64 bit file handle

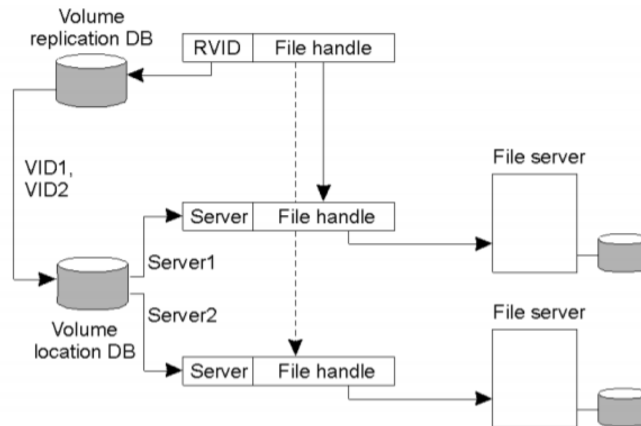


Figure 23.1: Coda architecture

23.1.5 Server Replication

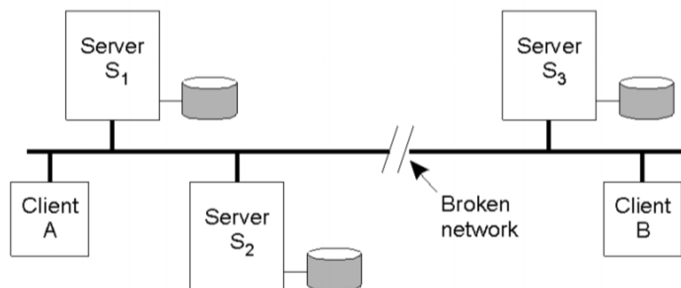


Figure 23.2: Server replication issues in Coda

- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG(all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2, 2, 1] and [1, 1, 2] is a conflict =>manual reconciliation

23.1.6 Disconnected Operation

- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection.

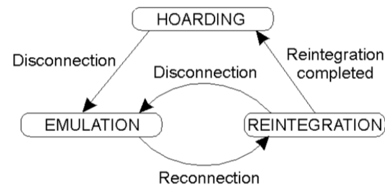


Figure 23.3: Disconnected operation in Coda

  - Prefetch all files that may be accessed and cache(hoard) locally
  - if AVSG=0, go to emulation mode and reintegrate upon reconnection

23.1.7 Transactional Semantics

- Notwork partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
    * Ensure that operations were serializable after thay have executed
  - Conflict => force manual reconciliation

23.1.8 Client Caching

- Cache consistency maintained using callbacks

23.2 xFS

23.2.1 Overview of xFS

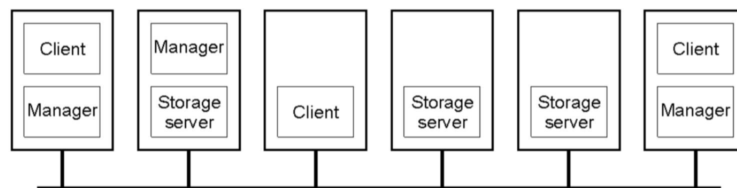


Figure 23.4: An example of nodes in xFS

- Key Idea: fully distributed file system [serverless file system]
  - Remove the bottleneck of a centralized system

- xFS: x in "xFS" => no server

- Designed for high-speed LAN environments

XFS combines two main concepts ; RAID - Redundant Array of Inexpensive Disks) and Log Structured File Systems (LFS). It uses a concept of Network Stripping and RAID over a network wherein, a file is partitioned into blocks and provided to different servers. These blocks are then made as a Software RAID file by computing a parity for each block which resides on a different machine.

In log structured File systems, data is sequentially written in the form of a log. The motivation for LFS would be the large memory caches used by the OS. Larger, the size of cache, more the number of cache hits due to reads, better will be the payoff due to the cache. The disk would be accessed only if there is a cache miss. Due to the this locality of access, mostly write requests would trickle to the disk. Hence, the disk traffic comes predominantly from write. In traditional hard drive disks, a disk head read or writes data . Hence, to read a block, a seeks needs to be done ie move the head to the right track on the disk.

How to optimize a file system which sees mostly write traffic ?

The basic insight is to reduce the time spent on seek and waiting for the required block to spin by. Every read/write request incurs a seek time and a rotational latency overhead. In general , random access layout is assumed for all blocks in the disk wherein the next block is present in an arbitrary location. This would require a seek time.

To eliminate this, a sequential form of writing facilitated by LFS can be used. The main idea of LFS is that we try to write all the blocks sequentially one after the other. Thus LFS essentially buffers the writes and writes them in contiguous blocks into segments in a log like fashion. This will dramatically improve the performance. Any new modification would be appended at the end of the current log and hence, overwriting is not allowed. Any LFS requires a garbage collection mechanism to de-fragment and clean holes in the log.

Hence, XFS ensures 1. fault tolerance - due to RAID, 2. Parallelism - due to blocks being sent to multiple nodes. 3. High Performance - due to Log structured organization.

In SSD's, the above mentioned optimization to log structures doesn't give any benefits since there are no moving parts and hence, no seek.

23.2.2 xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping
  -RAID = Redundant Array of Independent Disks

- Dynamically distribute control processing across all servers on a per-file granularity
  - Utilizes serverless management scheme.

- Eliminates central server caching using cooperative caching
  - Harvest portions of client memory as a large, global file cache.

23.2.3 Array Reliability

- Reliability of N disks = Reliability of I Disk N

  50, 000 Hours 70 disks = 700 hours

  Disk system MTTF:Drops from 6 years to I month!

- Arrays(without redundancy) too unreliable to be useful!

23.3 RAID

23.3.1 RAID Overview

- Basic idea: files are ”striped” across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - Capacity penalty to store redundant info
  - Bandwidth penalty to update redundant info

23.3.1.1 RAID

RAID stands for Redundant Array of Independent Disks. In RAID based storage, files are striped across multiple disks. Disk failures are to be handled explicitly in case of a RAID based storage. Fault tolerance is built through redundancy.

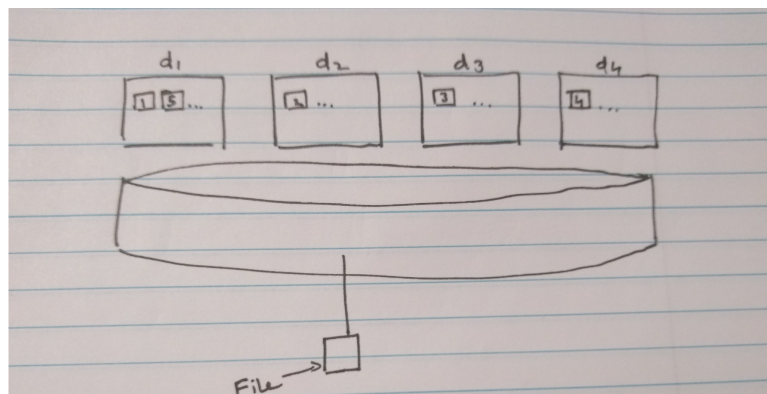


Figure 23.5: Striping in RAID

Figure 23.5 shows how files are stored in RAID. d1,...d4 are disks. Each file is divided into blocks and stored in the disks in a round robin fashion. So if a disk fails, all parts stored on that disk are lost. It has an advantage that file can be read in parallel because data is stored on multiple disks and they can be read at

the same time. Secondly, storage is load balanced. If a file is popular and is requested more often, the load is evenly balanced across nodes. This also results in higher throughput.

A disadvantage of striping is failure of disks. The performance of this system depends on the reliability of disks. A typical disk lasts for 50,000 hours which is also knows as the Disk MTTF. As we add disks to the system, the MTTF drops as disk failures are independent.

Reliability of N disks = Reliability of 1 disk $\div N$

We implement some form of redundancy in the system to avoid disadvantages caused by disk failures. Depending on the type of redundancy the system can be classified into different groups:

23.3.1.2 RAID 1 (Mirroring)

From figure 23.6, we can see that in RAID 1 each disk is fully duplicated. Each logical write involves two physical writes. This scheme is not cost effective as it involves a 100% capacity overhead.

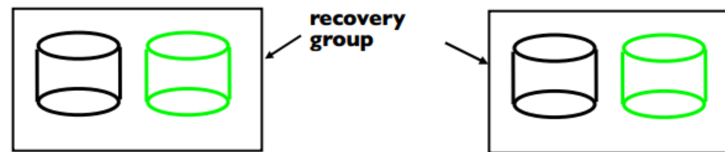


Figure 23.6: RAID 1

23.3.1.3 RAID 4

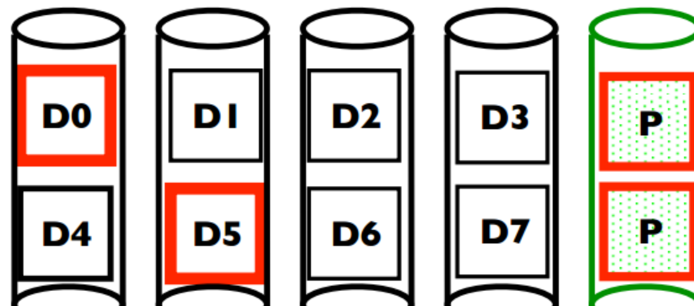


Figure 23.7: RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 23.7. First a parity block is constructed from the existing blocks. Suppose the blocks D_0, D_1, D_2 and D_3 are striped across 4 disks. A fifth block (parity block) is constructed as:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{23.1}$$

If any disk fails, then the corresponding block can be reconstructed using parity. For example:

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{23.2}$$

This error correcting scheme is one fault tolerant. Only one disk failure can be handled using RAID 4. The size of parity group should be tuned so as there is low chance of more than 1 disk failing in a single parity group.

23.3.1.4 RAID 5

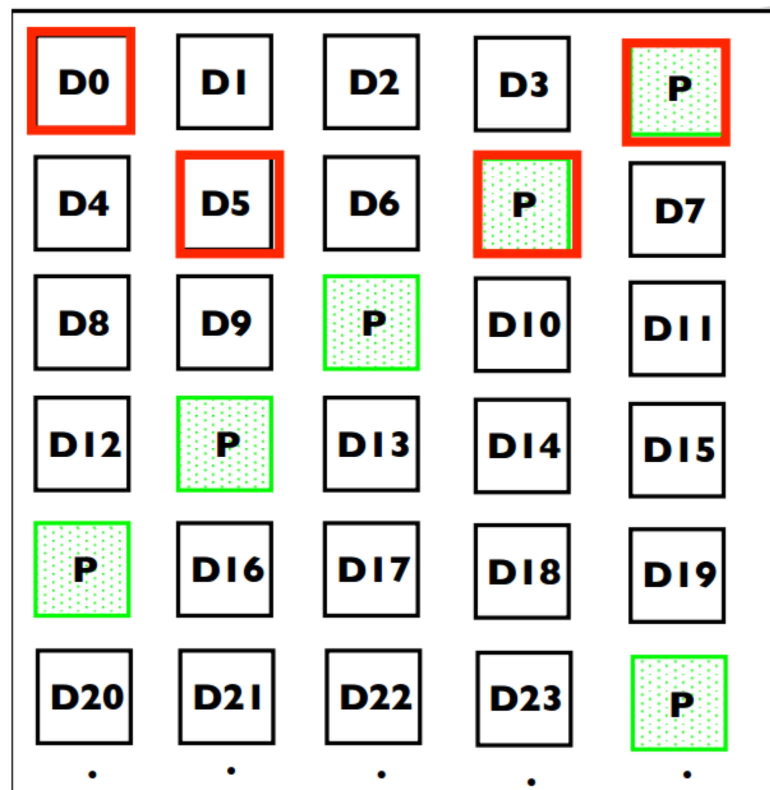


Figure 23.8: RAID 5

One of the main drawbacks of RAID 4 is that all parity blocks are stored on the same disk. Also, there are k + 1 I/O operations on each small write, where k is size of the parity block. Moreover, load on the parity disk is sum of load on other disks in the parity block. This will saturate the parity disk and slow down entire system.

In order to overcome this issue, RAID 5 uses distributed parity as shown in Figure 23.8. The parity blocks are distributed in an interleaved fashion.

Note: All RAID solutions have some write performance impact. There is no read performance impact.

RAID implementations are mostly on hardware level. Hardware RAID implementation are much faster than software RAID implementations.

23.3.2 xFS uses software RAID

- Two limitations

  - Overhead of parity management hurts performance for small writes
    * Ok, if overwriting all N-1 data blocks
    * Otherwise, must read old parity+data blocks to calculate new parity
    * Small writes are common in UNIX-like systems
  - Very expensive since hardware RAIDS add special hardware to compute parity

23.3.3 Log-structured FS

- Provide fast writes, simple revovery, flexible file location method
- Key Idea: buffer writes in memory and commit to disk in large, contiguous, fixed-size log segments
  - Complicates reads, since data can be anywhere
  - Use per-file inodes that move to the end of the log to handle reads
  - Uses in-memory imap to track mobile inodes
    * Periodically checkpoints imap to disk
    * Enables ”roll forward” failure recovery
  - Drawback: must clean ”holes” created by new writes

23.3.4 Combine LFS with Software RAID

Log written sequentially are chopped into blocks which a parity groups. Each parity group becomes a server on a different machine in a RAID fashion

23.4 HDFS - Hadoop Distributed File system

It is designed for high throughput - very large datasets. It optimizes the data for batch processing rather than interactive processing. HDFS has a simple coherency model in which it assumes a WORM (Write Once Read Many) model. In WORM, file do not change and changes are append-only.

23.4.1 Architecture

There are 2 kinds of nodes in HDFS ; Data and Meta-data nodes. Data nodes store the data whereas, meta-data keeps track of where the data is stored. Average block size in a file system is 4 KB. In HDFS, due to large datasets, block size is 64 MB.Replication of data prevents disk failures. Default replication factor in HDFS is 3.

23.5 GFS - Google File System

Master node acts as a meta-data server. It uses a file system tree to locate the chunks (GFS terminology for blocks). Each chunk is replicated on 3 nodes. Each chunk is stored as a file in Linux file system.

23.6 Object Storage Systems

- Use handles(e.g., HTTP) rather than files names
  - Location transparent and location independence
  - Separation of data from metadata
- No block storage: objects of varying sizes
- Uses
  \- Archival storage
  - can use internal data de-duplication

  \- Cloud Storage: Amazon S3 service
  - uses HTTP to put and get objects and delete
  - Bucket: objects belong to bucket/partitions name space

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 24: April 27

Lecturer: Prashant Shenoy *Scribe:* ***Avani Jindal***

24.1 Overview

In this lecture, we go over a few slides from the previous lecture covering HDFS (Hadoop Distributed File System) and GFS (Google File System). Then, we move to Distributed Middleware.

24.2 Previous Lecture

24.2.1 Combining LFS with Software RAID

The combination of disk array RAID and log-structured FS gives the notion of a serverless file system called xFS where any data can be stored anywhere. Files are stored as a log. Since we are using network striping, we take these logs and construct large and contiguous segments. Using these segments, we construct a parity and we stripe the entire log and the parity group.

24.2.2 HDFS : Hadoop Distributed File System

This file system is designed for high throughput access to data because it is used in distributed data processing. General purpose file system on our machines are designed for interactive access whereas this is designed for batch applications as distributed data processing is batch oriented. It has a simple model called the WORM (Write Once Read Many) model. We have large datasets and it is unlikely that we will change something after their creation. We might add to it or delete it, but we do not modify it again and again. Here, we try to move the computations to where the data is in order to reduce data movement.

Architecture: HDFS runs on top of a standard file system like the Linux file system. It is basically a layer which takes the file systems of individual machines and constructs a distributed file system out of it. The yellow boxes represent the individual machines and disks on them. The blue node is a metadata server. Some machines serve as metadata servers and they keep track of the location of data. To access a file, we first go to the metadata server and it gives the blocks of where the files are stored. We will get a list of n machines where n is the replication factor. We can go to any one and get that block of file. The green blocks are the replicated blocks. Two principles that we have to keep in mind - separation of data from metadata and that chunks of data are large & replicated on more machines.

24.2.3 GFS : Google File System

It is also designed for large scalability. There is a coordinator and a master node and chunk servers. The master node is the metadata server. It uses a replication factor of 3. It accesses a file in the way similar to HDFS. xFS, HDFS and GFS all have built-in fault tolerance. Here, the files are actually replicated because

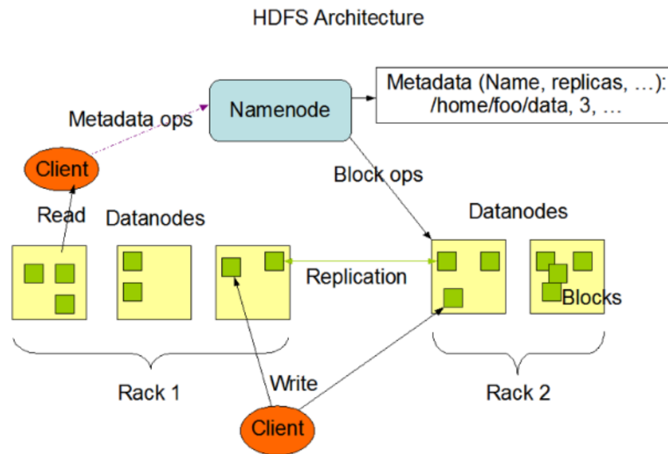


Figure 24.1: HDFS Architecture

we have large amounts of disk space and it is inexpensive to do so. Atomic writes means that when we update a file, we will create a version of it. The write will either go to all replicated files or none. The concepts used by HDFS and GFS are very similar.

24.2.4 Object Storage Systems

This is another way of storing large data files where the storage granularity is objects. The objects do not have a name like a file name, instead they have a handle. In most cases, the handle is actually a hash of the contents of the object. We have get and put operations instead of read and write operations respectively.

Question : What is meant by no block storage but object storage?
Answer : In block storage, the file system partitions the file into blocks and these blocks are stored on disk. In object storage, we can think of the object as being an entire block but there are varying sizes. We do not take an object and make it into smaller blocks and then spread those blocks out on a disk.

Question : What is the advantage of object storage over block storage?
Answer : The use case of object storage is archival storage and backups. We do not optimize object storage system for very heavy throughput applications, instead we optimize it for large amounts of data that we want to store but we are going to do infrequent reads and writes.

Question : Did object storage system evolve with cloud storage?
Answer : Object storage systems are newer than file storage. Cloud was essentially the first popularly available object storage system.

Now, we will switch to Distributed Middleware.

24.3 Distributed Objects

In case of remote objects, code on a client machine wants to invoke an object's method on a server machine. A common way to achieve this, is as follows. Clients have a stub called proxy with an interface matching the remote object. An invocation of a proxy's method is passed across the network to the 'skeleton' on the

server. That skeleton invokes the method on the remote object and returns the marshalled response. This can be recognized from earlier in the course as an RMI or RPC call.

Distributed objects are similar, but the distributed objects are themselves partitioned or replicated across different machines. Distributed objects use RPC. Middleware systems have been developed to support distributed objects.

24.4 Enterprise Java Beans

Enterprise Java is used to write multi-tier applications where the app server is actually written in Java. It also gives some additional functionality like the concept of a bean. A bean is a special type of an object. As a middleware, we will essentially have our objects written as a bean of some sort. It also provides other services like RMI, JNDI, JDBC(used to connect to Databases), JMS(Java Messaging Service). EJBs support more functionality that makes it easier to write web applications.

EJB are fundamentally object oriented, with two components, the interface and the implementation.

24.4.1 Four Types of EJBs

- **Stateless session beans -** They are essentially objects which do not have any state at all, they might just expose code.
- **Stateful session beans -** By default, the memory state is transient and if we kill the application, the object is gone. We can automatically persist the state of the object using these.
- **Entity beans -** They look more like standard Java objects.
- **Message-driven beans -** They are designed for messaging and the messages can persist.

24.5 CORBA : Common Object Request Broker Architecture

At the core of CORBA is the Object Request Broker (ORB) [also called messaging bus] is a intermediate communication channel that allows communication between objects.

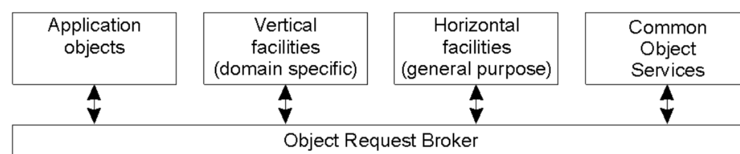


Figure 24.2: CORBA

The four boxes are the four components and they communicate using RPCs handled by the ORB. Many functionalities are already provided in CORBA as a service. They provide a dozen different services from concurrency to licensing in complex distributed systems. The advantage is that they can help reduce the code needed to develop complex distributed systems. However, in trying to provide every service you'd ever need, CORBA became very heavy weight. It does a lot of overhead to write even a small application. By becoming very heavy-weight, it became very difficult to learn and simple distributed applications would require deploying such a heavy weight system. Even though it did not become a commercial success, some

stripped down versions of CORBA actually got used. Example - messaging service in Linux desktop manager called gnome.

The stub in the object model of CORBA is called ORB. It uses Interface Definition Language (IDL) to use an interface and compiler to generate code (like protobufs). Proxy is used to specify the objects and services. Object adapter provides portability between languages. Thus CORBA, is language independent. CORBA provided even more flexibility having the option of invoking RPCs as any type including synchronous, one-way, or deferred synchronous. CORBA was one of the first distributed middleware systems. Modern middleware systems take many ideas from it.

24.5.1 Event and Notification Services

This is done using an event channel. It allows us to implement an application using the publisher-subscriber model. Publishers post events to the event channel, and consumers/subscribers ask for events that they subscribe to from the event channel. Publisher subscriber works with any combination of push and pull. In CORBA, it is a push-push model where data is pushed from publisher to event channel. The event channel will see the list of consumers subscribed and whenever there is a match, it will push to the consumer. In pull-pull model, event channel polls data from the publisher and similarly, consumer pulls data from the event channel.

24.5.2 Messaging

To implement all kinds of RPCs, CORBA has callback model. It will have a callback interface where we can register a callback function. Whenever a reply to the async RPC comes back, we are notified and we can get our reply. We can also use the polling method where we keep polling periodically to see if the reply has come back.

24.6 DCOM : Distributed Component Object Model

DCOM is Microsoft's middleware which has now evolved into .NET. COM is a simple RMI based framework running local to a machine that allowed communication within a machine. It is mainly used for communication between Microsoft applications. Object Linking and Embedding (OLE) was added to allow Microsoft office applications to communicate with one another via embedding and document linking. The ActiveX layer facilitates exposing these services as web applications by allowing us to embed things in web documents. Microsoft picked up this whole thing and made it distributed called DCOM. .NET has a language independent runtime, but ActiveX only works with Internet Explorer.

The architecture of DCOM is fundamentally the same as the distributed objects. The stub is essentially the COM layer. At its core, it is an RMI based system on objects. We also have a SCM (Service Control Manager) which keeps track of what all objects are in the system and where are they running.

The objects in DCOM can also be made persistent even though they are transient by default. It is done using the notion of a Moniker. Moniker is the name of a persistent object that allows us to reconstruct that object after we shut down the server application.

24.7 Distributed Coordination Middleware

In this case, we have a very loose coupling between how communication works. The idea is that we want to separate our computations from coordination. Distributed applications can be classified based on whats happening in time and space dimension. Applications can either be coupled or decoupled in space and time.

1. ⟨ coupled in space and time ⟩ Direct
2. ⟨ coupled in space but not time ⟩ Mailbox - receiver is known but receiver state does not matter.
3. ⟨ coupled in time but not space ⟩ Meeting Oriented - unknown who will show up to meeting.
4. ⟨ decoupled in space and time ⟩ Generative Communication - components can communicate with another without knowing who might read it or when it would be read. It uses a pub-sub model.

24.7.1 Jini Case Study

Jini is a Java based middleware that uses distributed coordination. It facilitates service discovery. We do not know what entities are present in the system so this notion of discovery allows us to discover what services are available and so on. These are also called zero-configuration services because we do not need to pre configure anything as services as discovered on the fly. It uses a event notification system that is pub-sub based. Jini uses a bulletin board architecture. Services advertise on the bulletin board and machines can access the services it requires through the board. It is decoupled in time and space.

In Jini, bulletin board is called JavaSpace or tuple space. Each tuple is a Java object. We have reads and writes into a shared database.

Question : Where is this JavaSpace running?
Answer : JavaSpace is our middleware service which has to run on some server or some set of servers. So essentially, our middleware is running somewhere else and we have to read and write from that.

Jini utilizes a pub-sub architecture with both pull based as well as notification based discovery.

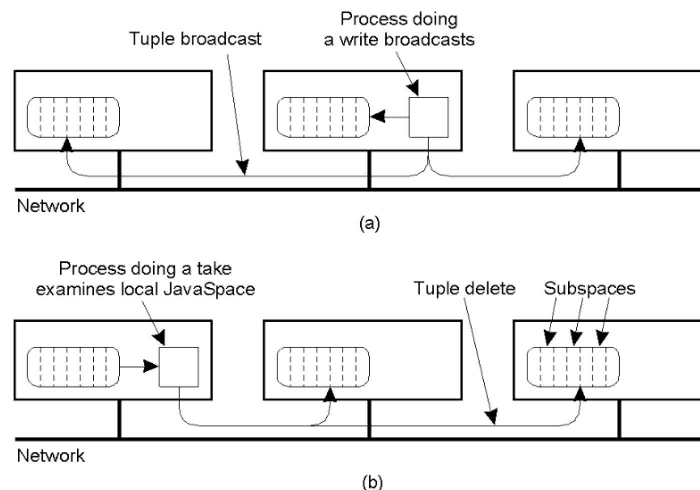


Figure 24.3: Jini Processes

Here, JavaSpace is distributed across multiple machines. The boxes are JavaSpaces with tuples in them. JavaSpaces can either be fully replicated or distributed. In case of replicated JavaSpaces, writes need to be broadcasted to all replicas whereas reads are local. On the other hand, in distributed bulletin board, each board has a subset of nodes, while writes are local and reads need to be done on each and every board.

Question : Whatever we are posting in the JavaSpace, is it a Java object and how is it posted (copied/sent)?
Answer : Tuples are not Java objects, they are data objects like a key and a value. So essentially, we are publishing data or events instead of objects which is different from sending/ receiving objects.

24.8 Big Data Applications

This is mostly covered in CS532 systems for data science. So we will only partially cover this.

Distributed data processing is different types of middleware that are designed for processing large amounts of data. The basic idea is that we will use multiple machines of a cluster and parallelize our application for data processing. Each machine will read and process some part of the data.

24.9 MapReduce Programming Model

MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware). Processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead.

MapReduce is a two-stage process to process a large dataset - Map phase and Reduce phase.

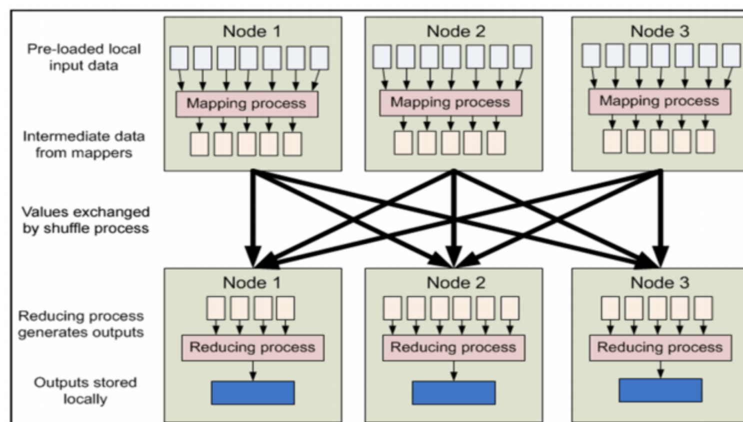


Figure 24.4: Map Reduce example

Map Step : Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of redundant input data is processed.

Shuffle Step : Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

Reduce Step : Worker nodes now process each group of output data, per key, in parallel.

Example: Let's say we have huge dataset of number (or may be words in a document) which need to be sorted (or frequency of words). Suppose we have multiple machines. Then each machine could take a chunk of data and sort it. Later, all these sorted chunks chould be merge together similar to merge sort algorithm. However, this requires huge communication between the machines during merge phase. To overcome this problem, let's say we know the range of numbers. In such a case, we could follow the bucket sort paradigm where each machine sorts a specific range of numbers. This way, inter-machine communication can be reduced.

The datasets being processed here are actually stored on HDFS. Each node can read its local data from HDFS or it can also read remote data. Reading locally has lesser overheads and is cheaper.

24.9.1 Other Programming Models

Apache Tez An application framework which allows for a complex directed-acyclic-graph of tasks for processing data.

Microsoft Naiad Naiad is system for data-parallel dataflow computation which attempts to raise the levels of abstraction used by programmers from an imperative sequence of MapReduce-style statements, to involve higher level concepts of loops and streaming.

Spark Apache Spark is a fast and general engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing. DAG with in memory resilient data sets.

Flink Apache Flink® is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications. DAG models extended to cyclic graphs.

24.10 Hadoop Big Data Platform

Hadoop is an implementation of Map-Reduce framework. It is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs. It has :

- **store managers** : where the datasets are stored. Eg: HDFS, HBASE, Kafka, etc (replication for fault tolerence.
- **processing framework** : Map-reduce, Spark, etc
- **resource managers** : allocates nodes and resources to jobs. Some of the concepts of distributed scheduling are also adopted since they need to serve multiple users. Example : Yarm, Mesos, etc

24.10.1 Ecosystem

Based on the requirements different types of frameworks can be used. For example, if user wants to process the data that have lot of graphs then Graph processing framework Giraph can be used. There are machine learning frameworks like MLLib, Oyyx, Tensorflow that are also designed to run to on Hadoop. If a user wants to input data to these distributed processing framework, he could use applications like hive to easily

write map-reduce codes. For real time data processing where data is generated continuously by some external source, framework like Spark Storm etc could be used.
We can see that it is not a single distributed application, it is a set of applications that work together.

24.11 Spark Platform

Apache Spark is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics. It was originally developed at UC Berkeley in 2009. Spark was an important innovation over MapReduce. Although MapReduce uses parallelism, it is very heavy on I/O that can slow down the application. In Spark, we store intermediate data in memory of some server. What we decide to store and how to store is something we have to think about when writing a Spark application.

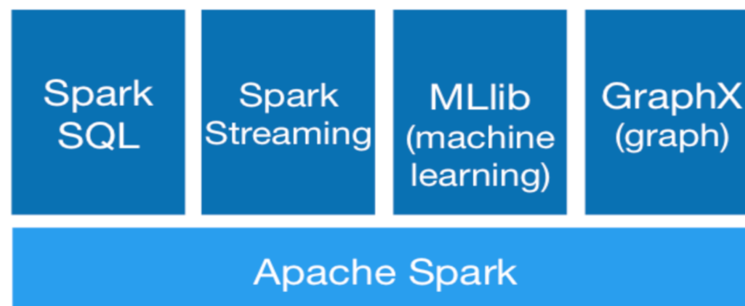


Figure 24.5: Spark Platform

Spark SQL : Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

Spark Streaming Many applications need the ability to process and analyze not only batch data, but also streams of new data in real-time. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter.

MLlib Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (up to 100x faster than MapReduce). The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

GraphX GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

Adavatages of Spark

- **Speed :** Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

- **Ease of Use:** Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.
- **Generality :** Combine SQL, streaming, and complex analytics.Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.
- **Runs Everywhere:** Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

24.11.1 Spark Architecture

Distributed memory is just like memory but we access data across various machines. All of the memories of the servers can be accessed if we store data in the form of an RDD. The idea is that we will first read data from disk, say HDFS. Then we will do some partial processing, then transformed dataset will be stored in RDD and so on. Since data is in memory, processing will be much faster.

RDD is designed to support in-memory data storage, distributed across a cluster in a manner that is demonstrably both fault-tolerant and efficient. Fault-tolerance is achieved, in part, by tracking the lineage of transformations applied to coarse-grained sets of data. Efficiency is achieved through parallelization of processing across multiple nodes in the cluster, and minimization of data replication between those nodes. Server failures can be handled by recomputation.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 25: May 2

Lecturer: Prashant Shenoy *Scribe:* ***Daniel Saunders, Arkin Dharawat, and Will Lee***

25.1 Distributed Systems Security

Three approaches to protect a system:

- Protect against invalid operations on your data.
- Protect against unauthorized invocations of your code.
- Protect against unauthorized users.

Security is more matter in a distributed system because the system is now accessible on a network. Other users can access it on top of authorized users.

25.1.1 Authentication

Idea is to prove to other party who you claim to be. Question is how do I know if the remote party is really who they claim they are.

25.1.1.1 Authentication Protocol (Ap)

- Ap 1.0: Simplest approach - one party/process is trying to prove to another their identity by sending a message but the problem is an intruder could also send such a message
- Ap 2.0: Another is to use IP address check and see if the party is coming from that known IP. Problem is IP spoofing where an attacker could insert a fake IP address into a packet and pretend to be coming the approved IP address
- Ap 3.0: Use usernames and passwords. Problem is attacker can look at the traffic and intercept the password through packet sniffing
- Ap 3.1: Instead of plain text, use encryption such as symmetric key.It uses a function to encrypt the message and without the right key, attacker cannot retrieve the message back.Receiver use the same key to decrypt the message. Problem is the re-play/playback attack. An attacker could intercept the encrypted message and hand it off to claim as the real sender. The receiving party could be fooled into decrypting the message to the attacker.

25.1.2 Authentication using Nonces

A nonce is an ”once-in-a-lifetime-only” integer generated for a session and will never be used again.

- Ap 4.0: The nonce is used by the receiver to challenge the sender to prove their identity. Receiver B will send the nonce to the sender. Sender will use a secret key to encrypt the nonce and send it back. When the receiver gets the encrypted message and if the decrypted message is the nonce, then the sender identify if verified.

Problem is it better be once in a lifetime so you don't pick it again. Make sure nonce is large enough in terms of bits.

25.1.3 Authentication using public keys

AP 4.0 uses symmetric keys for authenticantion. **Question**: can we use public keys? *symmetry*: DA(EA(n)) = EA(DA(n)).

AP 5.0:
Uses public key (EA) and private keys (DA)
A to B: msg = "I am A"
B to A: once in a lifetime value n
A to B: msg = DA(n)
B computes: If EA(DA(n)) = n
then A is verified
else A is fradulent

Problem: An attacker can intercept messages between sender and receiver and assume the identity of the sender. If the attacker can intercept and send its own public key when the receiver asks for it, the receiver would be tricked into decrypting the message using the attacker's public key. This is a problem of key distribution and without a secure way to distributed keys, public key encryption would not work.

25.1.4 Man-in-the-middle attack

Trudy impersonates as Alice to Bob and as Bob to Alice.

Alice		Trudy		Bob
	"I am A"		"I am A"	
			nonce n	
			DT(n)	
			send me ET	
	nonce n			
	DA(n)			
	send me EA			
	EA			

Bob sends data using ET, and Trudy decrypts and forwards it using EA (Trudy *transparently* intercepts every message).

25.1.5 Digital signatures using public keys

Goals of digital signatures:

- Sender cannot repudiate message never sent ("I never sent that").
- Receiver cannot fake a received message.

Suppose A wants B to "sign" a message M:

B can first encrypt a message with its private key - this is like signing a message and send the encrypted/signed message to A.A can then use B's public key to decrypt the message and see that it was indeed signed by B

B send DB(M) to A
A computes if EB(DM(A)) = M
then B has signed M

But it's not efficiency if a key is used to encrypt a large document.

Question: Can B plausibly deny having sent M?

25.1.6 Message digests

Encrypting and decrypting entire messages using digital signatures is computationally expensive. Routers routinely exchange data, which do not need encryption, but do require authentication and to verify that data hasn't changed.

A message digest is a compact summary/representational of the original message, like a checksum using a hash function. A hash function H converts a variable length string to a fixed length hash value. The user will then digitally sign $H(M)$, and send both M and $DA(H(M))$ which is the signature. The receiver can verify by taking a hash of the message and then take the sender's public key, extract the message, and compare them. This can verify who sent the message and that if the message has been altered.

Important property of H: We want to minimize any hash collision. In other words, we want H to have low collision probability. Given a digest x, it is infeasible to find a message y for which $H(y) = x$. Also, it is infeasible to find any two messages such that $H(x) = H(y)$ (hash collision). If the hash function H satisfies this property, then this scheme is much more efficient than digital signatures with public keys: we need only encrypt the fixed-length hash value $H(M)$, typically much shorter than M.

25.1.7 Hash functions: MD5

MD5 is no longer secure to use anymore

MD5 takes an arbitrarily-sized objects, splits it into smaller chunks, and hash each of the chunks. This method is recursed until we have a fixed-sized hash value.

25.1.8 Hash functions: SHA

MD5 is not secure anymore. Secure Hash Algorithms (SHA) hash functions:

- SHA-1: 160-bit function that resembles MD5.
- SHA-2: family of two hash functions (SHA-256 and SHA-512).

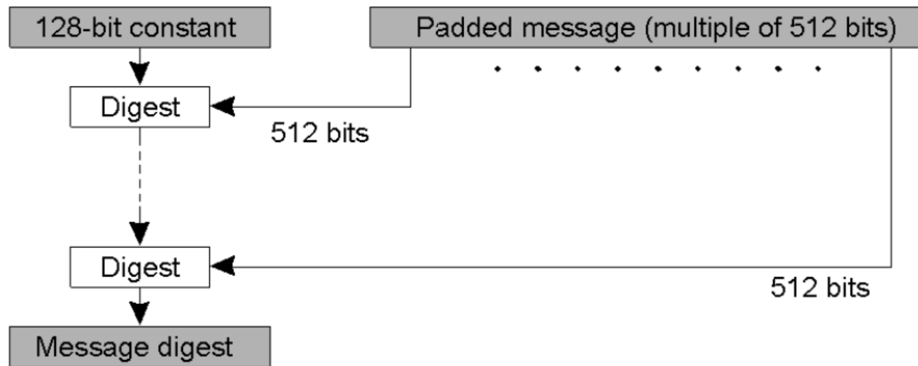


Figure 25.1: The structure of MD5.

- Developed by NIST and NSA.

Essentially, the larger the hash, the more secure but also more expensive to compute.Additionally, probabilities of collision and being attack are lower too

Let's say you have a hashed value h, and you want to know the original message m such that $H(m) = h$. How can we find out m? We could carry out a dictionary attack, in which we try all m' in some "dictionary" and compute $H(m')$. If any such m' produces $H(m') = h$, then $m' = m$. Clearly, the longer the message, the more time required by this brute-force attack.

25.1.9 Symmetric key exchange: trusted server

Problem: How do distributed entities agree on a key? How do you exchange keys securely?

This is the problem of key distribution. If keys are comprised, than anyone may use your key to decrypt your messages. We need a secure method of key exchange; it must be just as strong as your encryption algorithm itself.

Assume: Each entity has its own single key, which only it and a trusted server know.

Server:

- Will generate a one-time session key (symmetric) that A and B use to encrypt all communication.
- Will use A and B's single keys to communicate session key to A and B.

25.1.10 Key Exchange: Key Distribution Center

Trusted third party as the key distribution center. Alice sends a message to the KDC saying "I'm Alice, and I want to communicate with Bob; please generate a key". The KDC generates a random session key, encrypts it with Alice's public key, and sends it back to Alice. The KDC will send the same key to Bob, encryped with Bob's public key.

Public key crytography is only used to key distribution. Once keys are distributed, the sender and receiver would use the same key.

Question: Can an attacker intercept the generated key? An attacker can intercept these messages but would not be able to decrypt the messages to get the generated key.

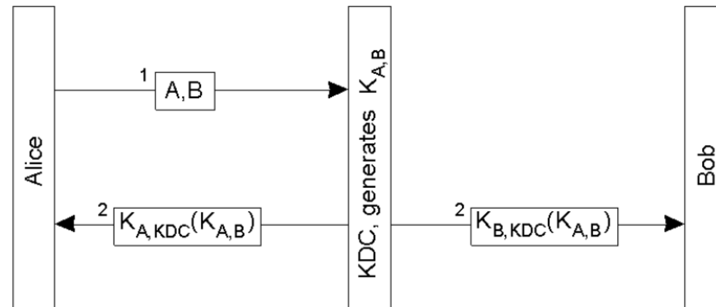


Figure 25.2: The principles of using a KDC.

25.1.11 Authentication using a key distribution center

Pictured below: Same technique as before, but the KDC does not actually send a message to Bob. Messages are sent from Alice to Bob.

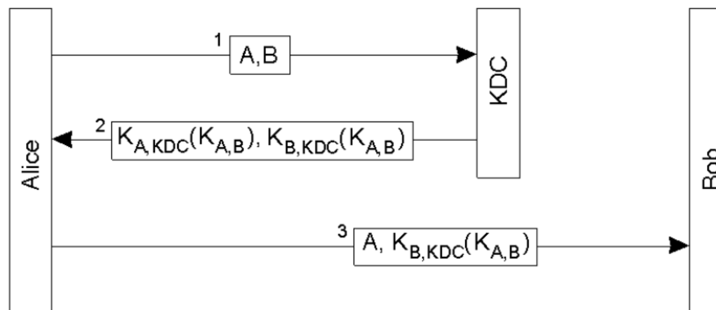


Figure 25.3: Using a ticket and letting Alice set up a connection to Bob.

25.1.12 Public Key Exchange

There is no KDC in this protocol; public and private keys are used only. This protocol assumes we have a secure way of getting someone's public key before communicating with them. Alice sends a nonce to Bob, which is encrypted by Alice with Bob's public key. Bob decrypts it, sends it back, sends a new nonce and session key back (all of this is encrypted with Alice's public key). Alice decrypts this message, and sends back Bob's generated nonce for validation purposes.

Public key retrieval is subject to man-in-the-middle attacks if public keys aren't shared securely. In other words, attackers can get hold of the actual public keys and replace them with theirs. We can use a trusted third party in order to distributed public keys securely. This third party will issue "certificates," public keys

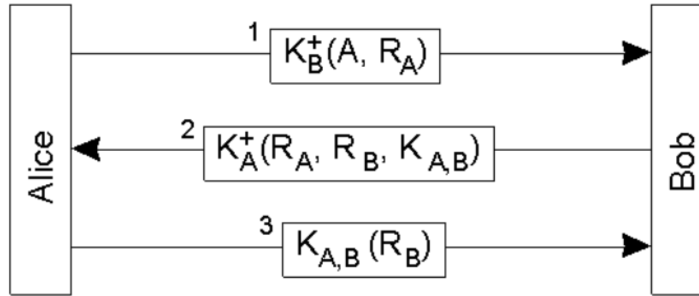


Figure 25.4: Mutual authentication in a public-key cryptosystem.

of servers which are signed by the trusted third party. Hence, when someone get your public key, it would come with a certificate with your name from a trusted certification authority and signed with its private key.

This is also how all web browsers communicate with the server using HTTPS. It would receive a certificate, verify it. Browsers have been pre-loaded with certifciate authority's public keys and could decrypt certificate for validation.

All servers have the trusted server's encryption key. Suppose that A wants to send B a message using B's "public" key; A can accomplish this by using certificates from the trusted third party. Certificates may be revoked.

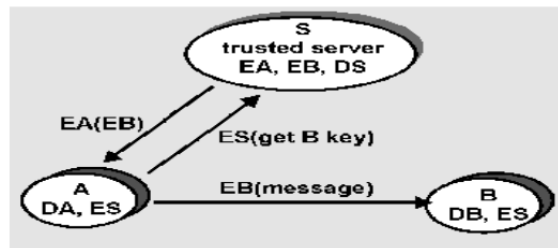


Figure 25.5: Public key exchange: trusted server.

25.1.13 Security in Enterprises

- In the multi-layered approach to security, Security functionality is spread across multiple components.
- Firewalls
- Deep packet inspection
- Virus and email scanning
- VLANS
- Network radius servers
- Securing WiFi
- VPNs

- Securing services using SSL, certificates, kerberos

Distributed systems security or network security are very complex problems. There are many layers of security protections that are deployed, and things get complicated fast. Companies typically get hacked because there is a small issue with one or more of the security items listed above.

25.1.14 Security in internet services

How can we secure internet services?

- Websites
  - Ensure all connections between browser and server are encrypted.
  - Authenticate user (username and password checks).
  - Use techniques like CAPTCHAs to prevent scripts from launching attacks.
- Challenge-response authentication
  - Push a one-time key to a user's phone to enter for further authentication.
- Two-factor authentication
  - Password and mobile phone (gmail).
- One-time passwords
- Online merchant payments: paypal, amazon payments, google checkouts

25.1.15 Firewalls

A firewall is a machine that sits on the boundary of the internal (e.g., home WiFi) and external networks (e.g., the Internet). All packets and traffic between the internal and external networks must pass through the firewall, which has a pre-configured set of packet-filtering rules. If a packet matches one of the firewall's rules, it is allowed to pass; otherwise, it is dropped.

Allow rules let the packet pass through. Deny rules block the packet from passing through. To block some traffic, one can set rules to deny packet from certain IP address.Rules can also set for outgoing traffic to deny access of certain web sites

- Firewall and Packet Filtering Firewalls: rules are used to either keep or drop packets. They don't look at contents. Only the headers of the packets.
- Application Gateways: high level firewalls look inside the packets (aka deep packet inspection) by intercepting packets for inspection to find potential virus signatures

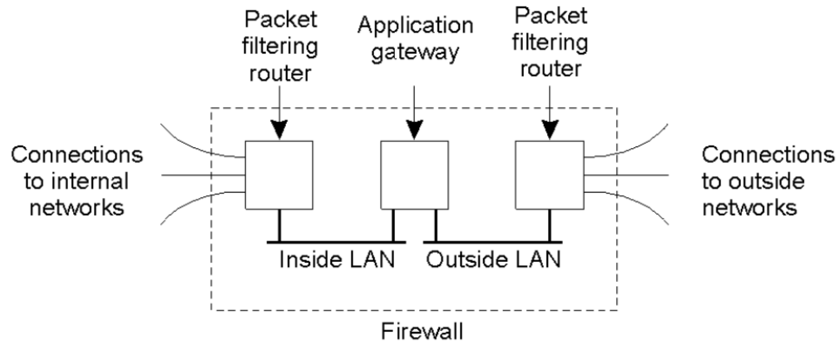


Figure 25.6: A common implementation of a firewall.

25.1.16 Secure email

- Requirements
  - Secrecy: No one else should be able to read your email other than the one intended
  - Sender authentication: Receipent should be able to verify who send the email
  - Message integrity: No one should be able to alter the content of the email
  - Receiver authentication: Only the appropriate receiver should be able to read the email
- Secrecy
  - Can use public keys to encrypt messages (this is inefficient for long messages because public keys are very long).
  - Use shorter, message-specific symmetric keys
    - Alice generates symmetric key K
    - Encrypt message M with K
    - Encrypt K with E_B
    - Send $K(M), E_B(K)$
    - Bob decrypts using his private key, gets K, and decrypts $K(M)$
- Authentication and Integrity (without secrecy)
  - Alice applies has function H to M (H can be MD5 or SHA)
  - Creates a digital signature $D_A(H(M))$
  - Send $M, D_A(H(M))$ to Bob
- Putting it all together
  - Compute $H(M)$, sign the hash $D_A(H(M))$ with the public key
  - M' = Concat the message and the hash: $\{M, D_A(H(M))\}$
  - Generate symmetric key K, compute $K(M')$
  - Encrypt K using public key as $E_B(K)$
  - Send $K(M'), E_B(K)$
- Used in PGP (pretty good privacy)

25.1.17 Secure sockets layer (SSL)

SSL (developed by NetScape) provides data encryption and authentication between web servers and clients. It lies above the transport layer. It is used for internet commerce, secure mail access (IMAP), and more. Features include: SLL server authentication, encrypted SSL sessions, and SSL client authentication.

Take a regular socket and add an encryption library on top. Send a message on the socket and the library will encrypt it. Socket on the other hand would then decrypt the message using the library.

It uses the HTTPS protocol instead of HTTP. The browsers sends the first message to the server saying it can support some version of SSL, and the server responds with its supported version of SSL, as well as a certificate (server's RSA public key encrypted by a trusted third party's (certification authority) private key). The browser generates a session key K, and encrypts the key with the public key E_S from the certificate authority. The browser sends "future messages will be encrypted" and $K(M)$ to the server, and the server responds with the same. The SSL session then begins.

Question: Are the keys store in the cookies of the clients? No. It's a session key and keep only in memory not on disk.Every time is connected, it will re-do for every socket.

25.2 Electronic payment systems

Payment systems based on direct payments between customer and merchant. Not the same as cryptocurrency, instead this is focused on using an electronic version of money instead of paper.

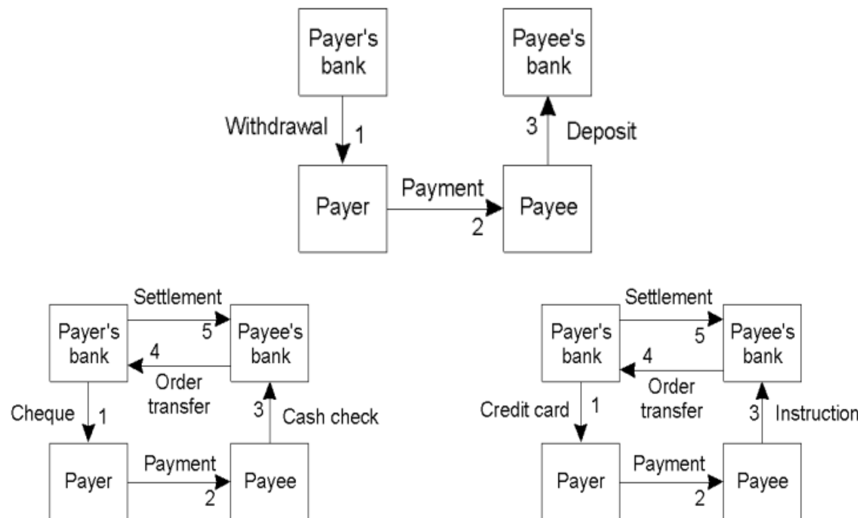


Figure 25.7: (a) Paying in cash; (b) Using a check; (c) Using a credit card.

25.2.1 E-cash

Paying in cash is inherently anonymous, Electronic cash aims to preserve anonymity of payments done similar to traditional cash.
This works on the principle of anonymous electronic cash using blind signatures. Users will first generate

a "coin" which will then be "blinded" (hide the sequence number of currency to the bank; this prevents tracking). This is sent to the bank to "sign" it. Afterwards the signed coin is unblinded, and subsequently given to a receiver as payment. The receiver can then check for validity with the bank, this involves checking for the signature and that the "coin" wasn't already spent. This prevent users from spending the same "coins" more than once.

Question: Would the bank need to authorize the pay? No since the payer withdraws the money and then generates the coin. There has to be an account with the bank but the bank only signs the the withdrawn money. This way the bank knows you withdrew some money but when the money is spent they can't know if its yours.

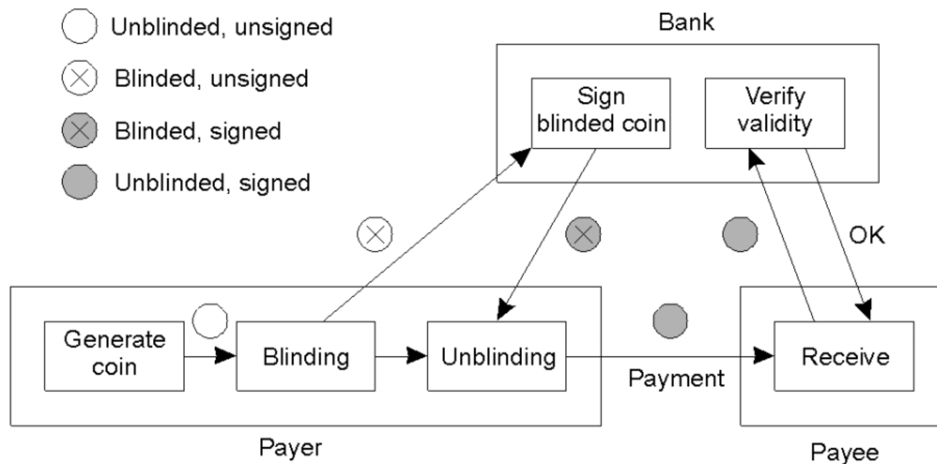


Figure 25.8: Illustration of E-cash.

25.2.2 Bitcoin

Cryptocurrencies are a form of digital currency which act as P2P electronic cash which are decentralized. Most popular one is Bitcoin, made by Satoshi Nakamoto based on Open source crypto protocol (proof of work etc.)
The currency in Bitcoin is generated by the participants (servers) in the network, so a peer-to-peer protocol is needed. The peers are responsible for validating transactions and performing other work for the network. All the transaction on the network are logged in a database called a Blochchain. The reward for the peers in participation is the new coins that are generated. The work to be done increases exponentially with each new coin generated, this is why from an environmental perspective this principle is not sound. Bitcoin uses digital signatures to pay "public keys" (receiver of the payment) and all these transactions are recorded on the blockchain.
User hold bitcoins in their digital wallets. If they want to pay someone they take some bitcoin (with a unique transaction number) and transfer it's ownership to the receiver. This transfer is recorded on a public record, without putting identity of payer/receiver on the ledger since the public keys are anonymous. Anonymity comes from this, that even though the owners are not known the transaction itself is public.

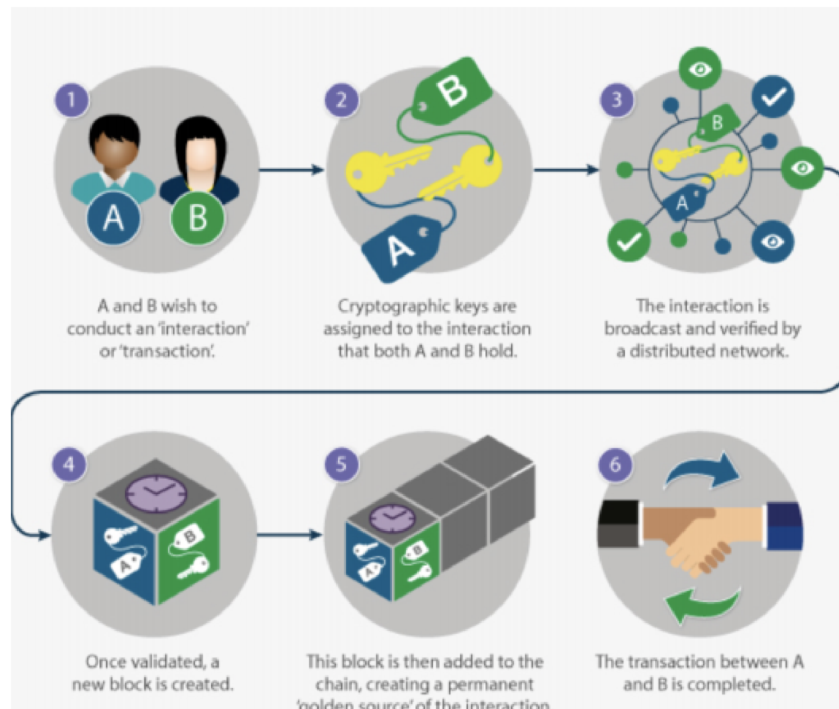


Figure 25.9: How Bitcoin works.

25.2.3 Blockchain: Distributed Ledger

Blockchain is a public distributed database which records every transactions that goes through the system. The transactions are not recorded based on identity but on basis of public keys, but every transaction is visible to all peers in the network. Thus it is acts as a public distributed ledger. It forms the basis of cryptocurrencies but can be applied in areas beyond cryptocurrencies too, e.g: stock register/trading, land purchases, smart contracts etc.
Any transactions made is signed with a private key and added into the ledger, and this makes it public. Every block consists of multiple transactions and once committed the transactions in the block are finalized. The blockchain is massively duplicated and shared using the P2P file transfer protocols. Special nodes called "miners" that append blocks to the global transaction record. Every time a new block is generated one of the peers appends it to the network and receives a reward, in case of bitcoin the "miners" receive bitcoin. All nodes of the network perform validation and clearing of transactions. Miner nodes perform "settlement" using a distributed consensus protocol.

Question: Did the concept of a blockchain come up with bitcoin? Blockchain was a separate concept, as a distributed decentralized database. Bitcoin made it practical and incentived people to actually participate in this decentralized network that implemented a distributed ledger.

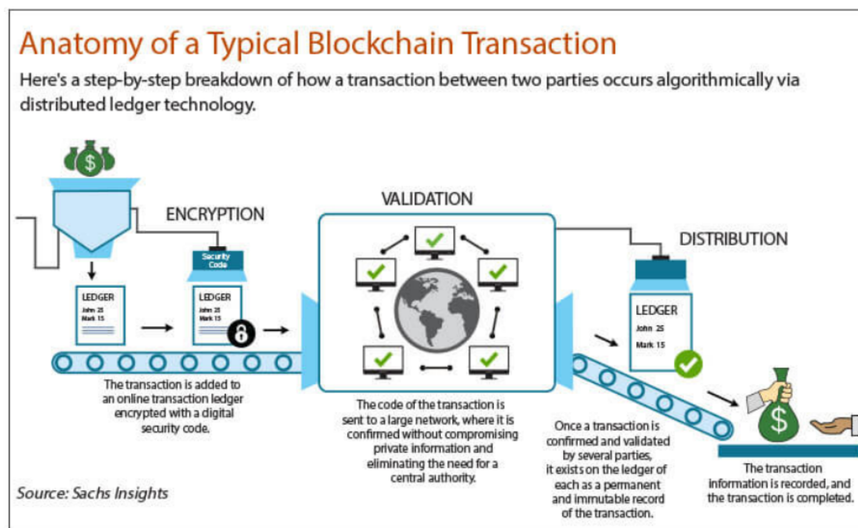


Figure 25.10: How blockchain works.

CMPSCI 677 Operating Systems **Spring 2022**

Lecture 26: May 4

Lecturer: ***Noman Bashir*** *Scribe:* ***Abhishek Pawar (2022), Suhas Keshavamurthy (2018)***

26.1 Introduction

This lecture covers topics related to Pervasive Computing, Internet of Things, Smart Buildings and Green Computing.

26.2 Pervasive Computing

Computing is becoming increasingly ubiquitous (sensing and computing is 'Everywhere'). Some popular examples of this include Smartphones, Smart buildings, Smart transportation, etc. For example : PIR sensor is used to detect when a person has entered a room and switch on the lights.

26.2.1 Rise of Pervasive Computing

The rise of pervasive Computing has been possible due to the below 2 aspects :

- Miniaturization of Computing : Sensors have become tiny in terms of physical dimensions. This is particularly true with the growth of MEMS (Microelectromechanical systems). MEMS sensor can transform mechanical, thermal, biological, chemical, optical, and magnetic phenomena into electrical signals. In future it is likely that sensor sizes will reduce to nano-scale dimensions.
- Internet of Things(IoT) : This is nothing but a network of physical devices which are able to talk to each other and then transfer data to the cloud.

26.2.2 Applications

26.2.2.1 Smart Health

Early wearable devices were able to track heart rate, number of steps, etc for the purpose of fitness tracking and monitoring sleep. Fitbit is the most popular example of such a wearable device. Newer technology do much more than this. For example: Smart Clothing - Devices embedded in the clothing are able to do on-body monitoring for sweat detection. Smart glasses are able to track gaze in order to detect fatigue.

26.2.2.2 Smart Buildings

Some of the examples of devices that make building infrastructure smarter include Thermostat, Smart Plugs, Smart appliances, Smart Lock etc. A Smart appliance like a smart-fridge is potentially able to detect food

spoilage. Nest Thermostats are examples of smart thermostats that can learn usage pattern and dynamically control temperature settings. These devices can be controlled through a phone or voice interface.

26.2.2.3 Smart Transportation

One of the major examples of Smart Transportation is *Connected Cars*. Potential applications include Accident avoidance (pedestrian detection etc.), fleet management (cars can start/stop at the same time) and real-time public transport alerts.

26.2.2.4 Smart Roadways

Sensors can be used to switch on street lighting only if an automobile is passing nearby or dynamically direct cars to different lanes. They can also be used to monitor road conditions and for the purposes of Traffic management. For Ex: Austin has a program where signals from Bluetooth-enabled cars are used to estimate traffic density On the streets.

26.2.3 Design of 'Smart Infrastructure'

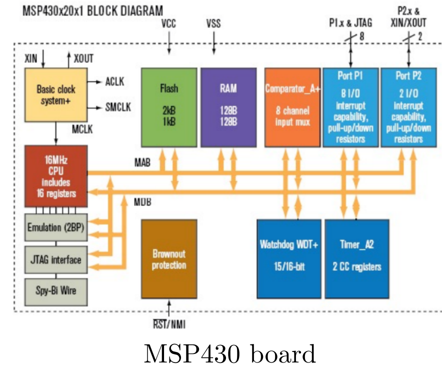
A typical application includes a personal device that is able to communicate with a mobile phone over a low-power network channel. The mobile phone then uploads the data to the cloud. In the cloud, the server performs analytics on the data and may provide feedback to the phone.

Another possible design is where the sensors in the environment upload data directly to the cloud. Ex: Smart Energy meters

Different elements of a distributed network of resource-constrained smart devices (also referred to as sensor nodes) are as follows -

26.2.3.1 CPU

The sensor nodes have very small CPU's. Popularly used CPU's are *Atmel AVR* (8-bit instruction set, 128 KB on-chip flash, 8 mA current draw) and *TI MSP430* (16-bit instruction set, 10 KB RAM, 48 KB flash, 2 mA current draw). As silicon technology evolves, the trend is for more compute to be available with lower energy consumption. Thus newer CPU's for these applications include *ARM 7* (32 bit, 50 MHz, 1 MB RAM), *ARM 9* etc



MSP430 board

26.2.3.2 Low-Power Radios

Communication is essential for a network of nodes to work together. The ISM Band (Industrial Scientific and Medical Band) is free for applications to communicate and is unlicensed. Some of the commonly used spectrum is 2400 MHz for Bluetooth and 900 MHz (33cm).
Different radio communication protocols work through different means. Zigbee (IEEE 802.15.4) works by modulating phase and Bluetooth (IEEE 802.15.1) works by modulating frequency of radio waves. These communication protocols typically have practical ranges of less than 100 meters. Chipcon CC2420 is a popular radio chipset.
An essential requirement of these radios is low power. Typically in these radios the energy consumption is higher while receiving than transmitting. This is because the radio needs to be 'ON' all the time, while it can transmit for a short time and go to sleep.

26.2.3.3 Battery Power

Battery Power is a big concern in IoT. For example: A Mica2 "mote" has a battery capacity of 2500 mAH. The mote consumption is around 25 mA which gives it a lifetime of 100 hours. In order to extend the life of the node, following are the possible alternatives :

- Bigger Battery : More capacity
- Energy Harvesting : Charge the battery in the node or use energy directly from the environment through sources like Solar, Wind or Motion
- Duty Cycling : Switch on and off the mote at a particular frequency based on the application

26.2.3.4 Sensors

The sensors in the system are responsible for measuring external phenomena. They can be used to measure temperature, humidity, light, acoustics, location etc. For example : the phone screen brightness is adjusted based on ambient light.
Sensor fusion is the technique of deriving insights by fusing data from multiple sensors. Another challenge is sensor placement. The multiple sensors need to be placed such that they can cover the entire area that needs to be monitored and provide redundancies in case of failure. Apart from the use of batteries, it is possible for sensors to harvest energy from the environment to power themselves.

26.2.4 Typical Design Issues

The typical concerns while building a pervasive computing systems are :

- Node : Maximize the lifetime of the node by increasing the battery power through various techniques.
- Network of Sensors : The data from various kinds of sensors need to be aggregated together. Localization and synchronization among the nodes in the network based on signal strength, direction, etc is also important. Another concern is the protocol for routing the communication from one node to another.
- Server Side Processing : On the server, we can employ big-data analytics in order to derive insights from the data and provide recommendations and alerts.

26.3 Green Computing

There are two major aspects of Green Computing

- Greening of Computing : Here the attempt is to design energy-efficient hardware, software and systems.
- Computing for Greening : It is the use of computers to make physical infrastructure more efficient.

26.3.1 Historical Overview

Historically the necessity for better energy efficiency has been driven by the need for energy-efficient mobile devices where the motivation has been longer battery life, and growth of data center where the motivation has been to lower the electricity costs. Green computing is also important in order to lower the carbon footprint and to improve efficiency of other systems. The typical *compute* energy consumption is 20% in an office building, 50-80% at a large university. It is 3% of the total energy consumption today and is growing rapidly.

26.3.2 Data Centers

A data center is a facility for housing a large number of servers and data. For example: A Google data center can be many football fields big, housing more than 100000 servers and consuming 100 MW of electricity. The energy cost of running a data center is very high. PUE or Power Usage Effectiveness is the ratio of total energy consumed by a data center to the energy delivered for compute. Typical PUE is between 1.5 - 2. A PUE of 2 indicates that the energy cost to run the data center is twice that of delivering power to the servers. Google achieves a PUE of 1.1 which is an indication of high efficiency of Google data center infrastructure.

Data Center

26.3.3 Greening of Computing

Following are the considerations while designing computing systems that are more energy efficient.

- Reduce Server Cost : This is done by purchasing more energy-efficient hardware including servers, power supplies etc. In certain cases it is more efficient to run the data center through DC rather than AC.
- Server Management : The power cost can be reduced by intelligent power management of servers by turning them off while not in use; use of virtualization by moving applications to a concentrated number of physical machines running virtual instances.
- Cooling Cost : This would involve design of better air-conditioning, thermal engineering and innovative methods of cooling like moving data-centers to cold places like Iceland etc.
- Desktop management : In large companies with thousands of desktops, IT tasks like update, virus scan are generally scheduled during the nighttime. Automatic sleep and wake up policies enable better desktop power management and reduce overall power consumption.

26.3.4 IT for Greening

Modern buildings have a distributed system of nodes. They are used to perform the following actions

- Monitoring by Sensors : Energy consumption in the building can be monitored at the outlet-level or meter-level.
- Analyzing the data : Real-time usage data is provided by the energy monitors/sensors to the Building Management System (BMS). This data can be used for modeling, analytics and prediction using statistical techniques and ML. In a residential setting, smart thermostats like *Nest* can learn household patterns by collecting energy usage and temperature data over a period of time.
- Use of Renewables : There has been a significant growth of renewable energy adoption including Rooftop wind turbines, solar PV installation and solar thermal power. One of the challenges in the usage of renewable energy is that it is intermittent in nature and is influenced by rapidly changing external environmental conditions like cloud cover, temperature etc.
- Forecasting of renewable energy : The challenge is to design predictive analytics in order to model and forecast energy generation. One of the use case of forecasting is in EV charging stations. Solar panels are installed in parking lots, rest areas and garages. Intelligence is required to design charging schedules to address questions like *when to charge ? Which EV to charge ? and by How much ?*

The data can also be used to motivate consumers to be more energy efficient as well as reveal insights into usage patterns, cause of wastage and improve efficiency.