Distributed and Operating Systems
Course Notes

Prashant Shenoy
University of Massachusetts Amherst
Note: The notes may contain some errors, we are working on fixing the mistakes

Contents

Lecture 1:Introduction

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 2nd and 4th editions of "Distributed Systems" by Tannenbaum and Van Steen as the textbook. It is legally available for free. In fact, all download links are on the **course material** section of the course website.

The instructor also introduced the grading scheme and other resources. It is worth noting that three programming assignments (45%) and two exams (one midterm and one final) (44%) contribute heavily towards the final grade. The course also includes lablets and homework problem sets for 10% of the grade and 1% for class participation. 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. All three sections do the same work and have the same grading policy.
2. Remote and in-class students can create a project group together.

Course Outline:

- Introduction - Basics, What, Why, Why not?
- Distributed Architectures
- Interprocess Communication - RPCs, RMI, message and stream-oriented communication
- Processes and scheduling - Thread/processes scheduling, code/process migration, virtualization
- Naming and location management - Entities, addresses, access points
- Canonical problems and solutions - Mutual exclusion, leader election, clock synchronization, etc.
- Resource sharing, replication, consistency - DFS, Consistency issues, caching, replication
- Fault-tolerance
- Security in Distributed Systems
- Distributed Middleware
- Advanced topics - Web computing, Cloud computing, edge computing, sustainable computing, big data, multimedia, Internet of Things (IoT)

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 a standalone system. Most of the online applications that we use on a daily basis are distributed in some shape or form. Examples include the 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 and this course will cover the principles and how to build these large-scale systems.

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

Definition: Broad definitions can be given.

1. **Networked System:** Multiple CPUs/Computers connected together over a network.
2. **Distributed System:** A networked collection of multiple computers that appears to its users as a single coherent system.

Decentralized Systems are networked systems where processes/resources are *necessarily* spread over multiple computers. In the case of **distributed systems**, the processes and resources are *sufficiently* spread across multiple computers with added distribution logic like replication and transparency.

Question: Is decentralized systems a subset of distributed systems?

Answer: No, decentralized systems are a superset of distributed systems. All distributed systems are decentralized but not every decentralized system is a distributed system.

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

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 of 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 backup server (failure transparency). Although incorporating all the transparency features reduces complexity for users, it also adds complexity to the system. Overall, a good design principle is to hide needless complexities from the users and only expose simpler abstractions. This is so that it is easier for the users to access and use the system.

Here are some transparencies:

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

- **Replication.** User can access the resource without knowing that it is replicated.

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

Question: Why is it called *transparent* when in fact we are hiding it from the user?

Answer: Transparency is a technical term that has been used to mean hidden from the user and should not be confused with the usual definition of the word.

Question: Is it still failure transparency when one server goes down and the request is routed to another server, but the second server gets overloaded and fails?

Answer: Yes. Failure transparency and routing the request to the second server is failure transparent as the user is unaware of the failure of the first server. To provide a good experience to the user, the design of the system should be such that the requests do not fail or slow down when the second server gets overloaded.

Question (previous year): When searching on Google, does the response time that shows how long the query was processed imply replication?

Answer: No. The response time does not tell us the degree of replication. It is considered as *replication transparent* since you cannot interact with a particular replication (replication is hidden from the users).

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.

It is worth noting that the term *open distributed system* is not the same as the term *open source system*. The term *open source system* means that the source code for the system is available, and users can download the source code and use it.

1.6 Scalability Problems and Techniques

It is often hard to distribute everything you have in the system. There are three common types of bottlenecks 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 about the system (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.** In other words, no machine should know what happens on all machines at all times. Here, a state can be thought of as data, file, or information.
2. **Algorithms should make decisions based on 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 a 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 an incremental build system) but you should not assume it is perfectly synchronized across all machines.

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

Question: In a distributed system, can a load balancer be a single point of failure?

Answer: Yes, for a distributed system with a single machine load balancer it can be the single point of failure. Therefore need to either replicate the load balancer or have another machine willing to take on the task.

Question (previous year): If you are to make a decision based on local information without asking other nodes to provide information or using global information can you make an incorrect decision?

Answer: Yes. For example, load balancing amongst three servers. Amongst the three servers, we want to send a request to the least-loaded server. If we ask all the servers for their current load before sending the request, we are using the global information, which affects scalability. If we are making a decision based on local information, we can do a round-robin. This method might not give us the most optimal solution, but it provides us with low overhead.

Question (previous year): If you want to make a centralized algorithm distributed, are you going to run the same code on multiple machines?

Answer: No, that is replication. The distributed algorithm would mean that you need to come up with a different way of implementing the algorithm.

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 into a machine and the machine gets overloaded. The machine wants to offload some tasks 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 machines about global knowledge, this is a much more scalable algorithm. A simple algorithm can be a random algorithm where the machine randomly picks 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 a 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 is 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 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 that the server is now distributed in location and is connected over a 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 and data center

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 microcontrollers 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 such as file system, memory manager, process manager, etc. are implemented as standard processes in userspace.

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

Answer (Student): Inter-process message communication between OS modules becomes a bottleneck and gives slower performance than function calls.

Answer (Instructor cont.): There is a lot of communication that has to happen between all modules for an OS to achieve its task. In other words, there is a performance penalty incurred from inter-process communications. For example, when you start a new process, you must send a message to the memory module requesting RAM.

Hybrid architecture: Some functionalities are independent processes while other functionalities are moved back to the kernel for better performance while having the modularity benefit from a software engineering perspective.

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. It provides transparency in terms of location, migration, concurrency, replication, etc.

There are essentially three flavors of distributed OS: 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 that 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 the user. Middleware takes a NOS and adds software on the top of the network OS services layer to make it behave or look like a DOS.

1.8.3 Multiprocessor Operating Systems

The multiprocessor operating system is like a uniprocessor operating system. It manages multiple CPUs transparently to the user and each processor has its own hardware cache, in which the consistency of cached data needs to be maintained.

Question: In the distributed OS, are the kernels communicating with each other over the network?

Answer: Yes, the distributed services are responsible for managing the resources on multiple machines.

Question: Will distributed OS make it difficult to add a machine to the network?

Answer: To some degree. You have to coordinate with the OS system service because you are adding another machine. As opposed to the network OS, you just put another machine to the network and you are done.

Lecture 2:Architecture Styles

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

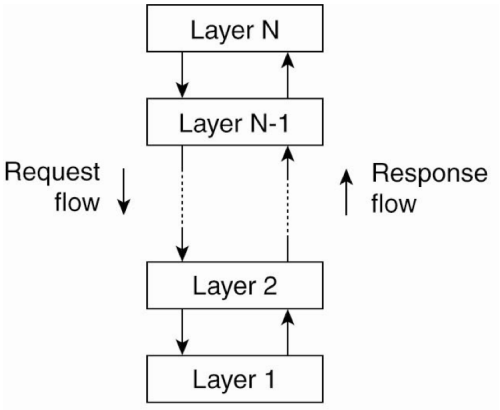


Figure 2.2: Layered Design

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 with 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 stacks. A 3-tiered web application typically would have the HTTP/Web Server, Middle Tier: Application Logic, Third Tier: Database

2.1.2 Object-Based Style

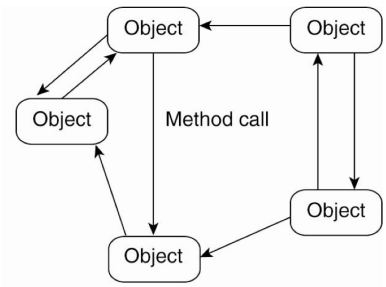


Figure 2.3: Object-based Style

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 has its state and exposes its 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

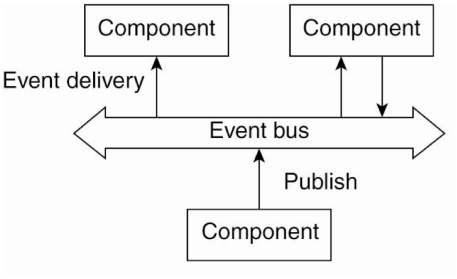


Figure 2.4: Event-based Architecture

An event-based architecture has many components that communicate using a publisher-subscriber (pub-sub) model via an event bus instead of 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 are found, the event bus will deliver the data to the appropriate component(s). There are many kinds of event buses, e.g., memory-based or disk-based.

Question (Student): Would the publisher and subscriber have different data structures? **Answer (Instructor):** The event that's being published and consumed would have a defined structure, but the event's mapping to internal data structures is irrelevant, and the publisher/subscriber can have their own representation internally.

Question (Student): Is the implementation of an event bus like a queue? **Answer (Instructor):** Queue based implementations(Message Queuing systems) are common for designing systems like this. A new event is added to the queue, and its recipients can be notified about this event. Ex of messaging queuing systems: RabbitMQ

Question (Student): Can you do 1:1 communication in Pub/Sub system? **Answer (Instructor):** In Pub/Sub systems, when an event is produced, there is no recipient to which it is addressed. The event bus delivers these events to the subscribers. We can emulate the 1:1 behavior by having one producer and subscriber, but it's not technically 1:1 communication.

Question (Student): Can the event bus be a single point of failure? **Answer (Instructor):** The system can be designed in a way that it's reliable and not a single point of failure.

Question (Student): Is the event bus access transparent? **Answer (Instructor):** Components are not aware of the events being handled inside.

2.1.4 Shared Data Space

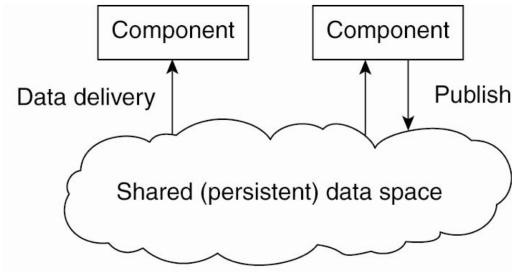


Figure 2.5: Shared Style

The shared data space architecture has a shared data space which is like a physical bulletin board. A component posts information and some components 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 retrieves 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 received in real time.

Question (Student): What does persistent mean in this context? **Answer (Instructor):** Persistent here means that the data is going to reside in the shared data space for an arbitrary amount of time. Technically, this means data is stored on disk.

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 applications as resources makes 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): Is it secure to use HTTP for RoA? **Answer (Instructor):** You can use HTTPs to secure the communication which is built over HTTP.

Question (Student): Is the difference between resource-oriented and object-oriented just that the former uses HTTP? **Answer (Instructor):** Yes, that's precisely it. RoA requires that the objects communicate via HTTP with a standard naming scheme(URL).

Question (Student): What does it mean to say messages should be fully described? **Answer (Instructor):** This architecture does not keep any state(state-less), so every request needs to contain all the information needed for it to be complete.

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.

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): Are object replies cacheable? **Answer (Instructor):** In RoA and SoA since the responses are HTTP or web, they are cacheable. OOA is not designed for caching.

Question (Student): How does cacheing rely on protocol? **Answer (Instructor):** Cache can be built for OOA, but it's not designed for it, but for the web, it comes as an additional feature.

Question (Student): What does marshalling parameters mean? **Answer (Instructor):** When communication is done over a network, the parameters must be sent in a standardized fashion to handle compatibility issues. Converting data to this standardized form is called marshaling.

Question (Student): Is resource oriented a superset of service-oriented? **Answer (Instructor):** There are different flavours of architectures.

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 resp. 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.

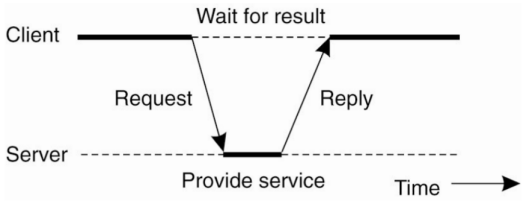


Figure 2.6: Client-Server Architecture

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

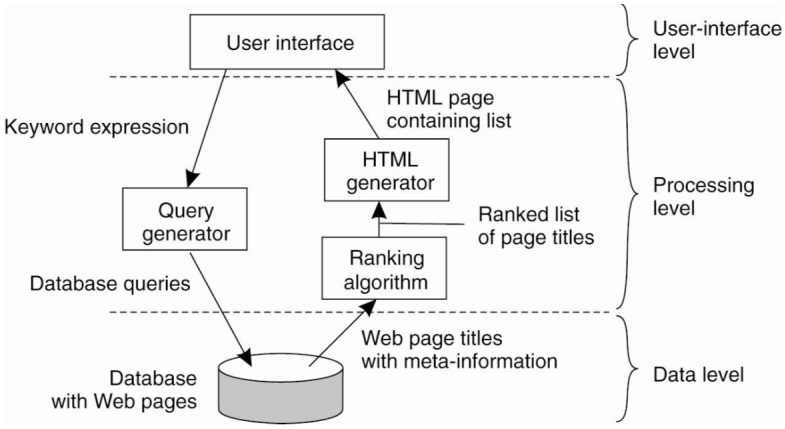


Figure 2.7: Search Engine Example

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 which responds with relevant results. 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 HTML generator then generates the page and is 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.

Question (Student): Where would the caching be done for commonly used queries? **Answer (Instructor):** It's typically in front of the database layer. On a cache miss, the database layer computes the query.

2.2.2 Multitiered Architectures

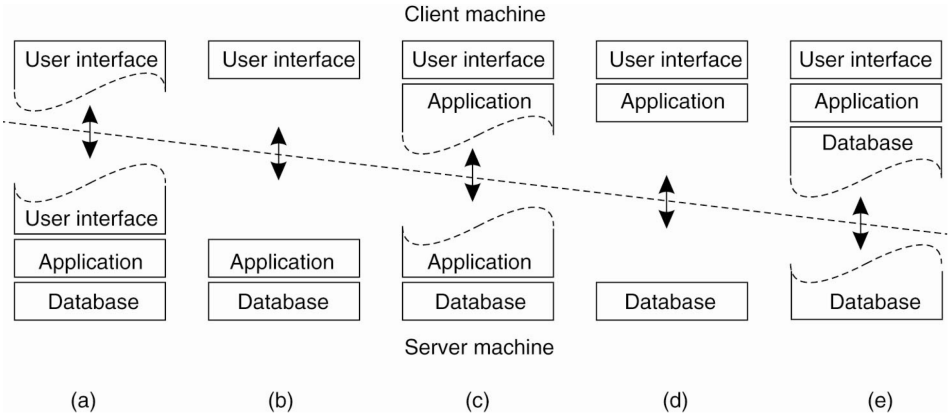


Figure 2.8: Client Server Choices

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 a 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.

Question (Student): Is the (e) in the above figure a cached application? **Answer (Instructor):** It could be cached where you have a database cache that stores something, or it could even be that some part of the database is on the client and some part of the database on the server.

2.2.3 Three-tier Web Application

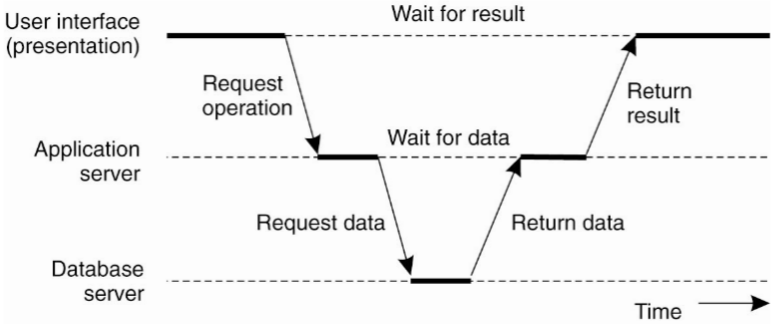


Figure 2.9: Three-tier Web Application

The three-tier web application architecture is a very popular architecture choice. It's an example of the layered architecture discussed above. 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 different depending on the context as we saw in the previous section.

Question (Student): Does every spectrum from (a)-(e) in fig 2.8 follows the working mentioned in fig 2.9? **Answer (Instructor):** This working shows how request flows and processing is done. According to system, it could be possible that it does not go through all the tiers.

2.2.4 Edge-Server Systems

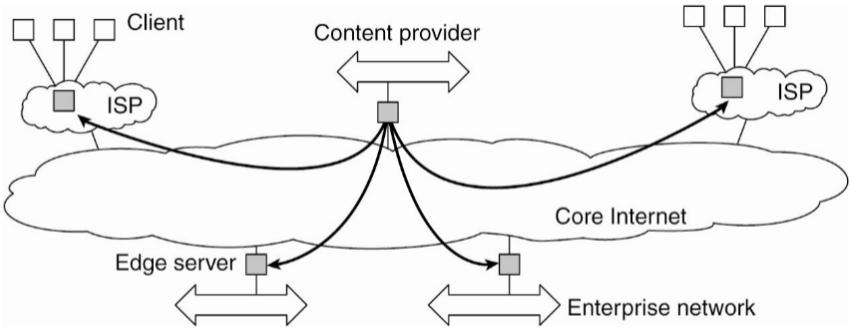


Figure 2.10: Edge Server

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.

Question (Student): Is the edge server a replica of the main server or runs a subset of the functionality of the main server? **Answer (Instructor):** Either can be true, it's an application choice.

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. Ex: Bit torrent system for sharing files.

We will introduce a structured peer-to-peer system named "Chord" as an example.

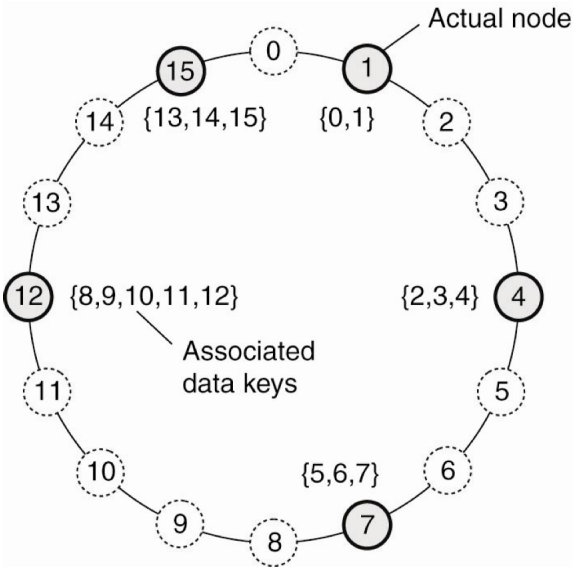


Figure 2.11: Chord Structure

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 happens when a peer goes offline? **Answer (Instructor):** In a client-server architecture, we assume the server is reliable, but here, the peer is inherently unreliable. If a node goes down, the system needs to handle repartitioning the responsibility of file owners.

2.3.1 Content Addressable Network (CAN)

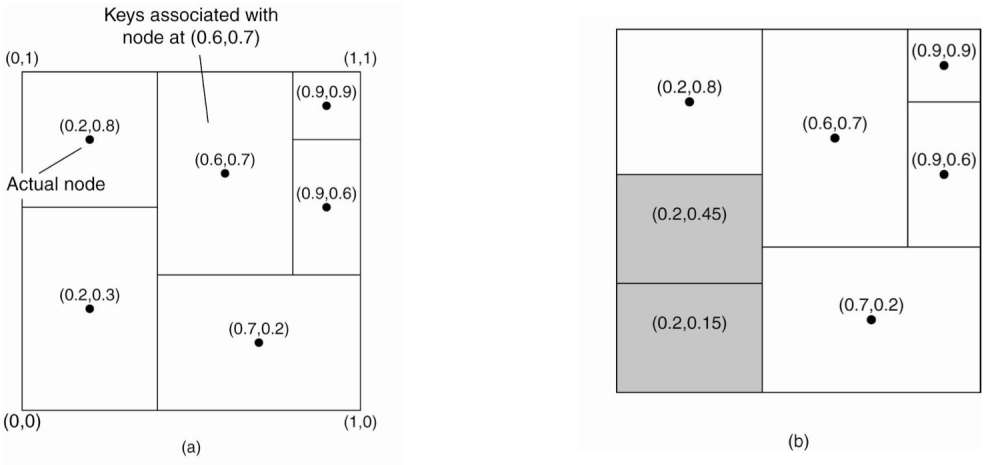


Figure 2.12: CAN Structure, with (b) showing a join procedure

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.

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.

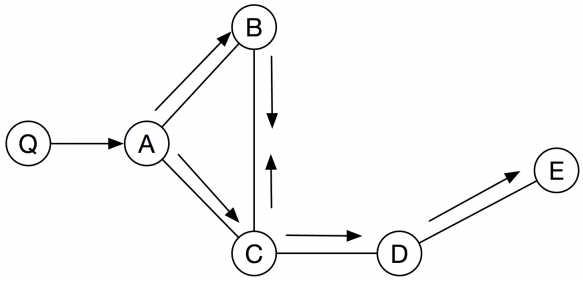


Figure 2.13: Search in Unstructured P2P System

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

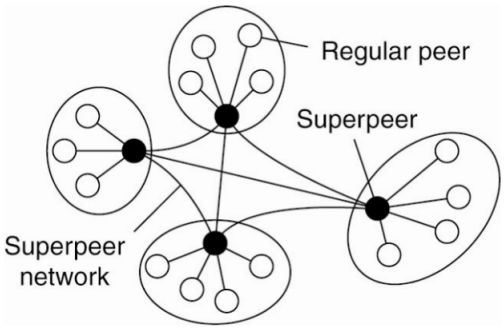


Figure 2.15: Graph with SuperPeer Structure

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. Superpeer is responsible when the peer within the cluster wants to communicate with other peers with different superpeers. 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.

Lecture 3:Decentralized Architectures

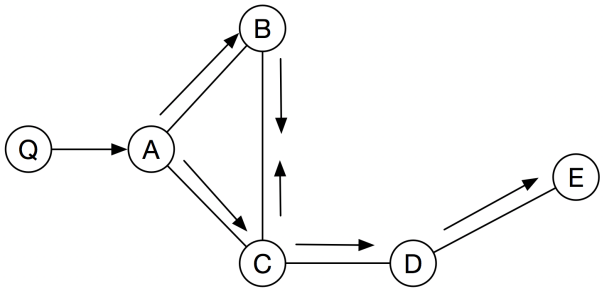
3.1 Decentralized Architectures (Module 3) Contd

3.1.1 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 3.1: 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.

Question (Student): Is there a way to decide which node to inject the query to?

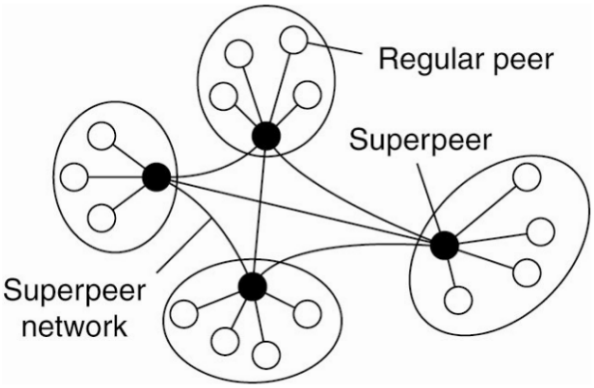
Answer (Instructor): The assumption is that the users are making these queries. So the user will run a copy of the peer-to-peer application on thir machine which is a node in the network. So, the user injects the query at your local peer and then it propagates to the rest of the network.

Question (Student): Can this query search create a cycle?

Answer (Instructor): Network topology can be arbitrary so the query might propagate and come back to the same node. But if nodes keep track of the queries already seen, the cycle can be avoided. Keeping track of outstanding queries solves the problem because nodes wait for the responses to come back in the reverse direction.

3.1.2 SuperPeers

Figure 3.2: 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.

Question (Student): What happens if the query doesn't reach the node that has the content because the number of hops a query can traverse is set to a small number?

Answer (Instructor): It's possible the content won't be found since the query distance is not sufficient. But generally, in a peer-to-peer network where nodes come and go, multiple copies of the content are present in the network. Thus by random chance, with a reasonable threshold, one of the nodes having the content can be reached. Hence, its probabilistic with no guarantees.

Question (Student): Is there an advantage to flooding versus having a list of nodes and then asking these nodes in some order?

Answer (Instructor): Since it is a peer-to-peer network, it's a decentralized network. There is no central entity that keeps track of all the nodes.

Question (Student): What happens if the superpeer is down?

Answer (Instructor): A new super peer is chosen using leader election.

Question (Student): Can you have another layer on top of super peers where they elect another set of peers that are even more distinguished than the super peers?

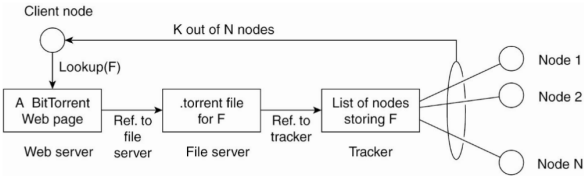
Answer (Instructor): This architecture is also possible. But this can result in a degree of centralization that should be avoided in a decentralized system.

Question (Student): If a new super peer gets elected, how do other nodes know about it?

Answer (Instructor): There should be a protocol by which the super peer advertises its election as a super peer and connects to other super peers.

3.1.3 Collaborative Distributed Systems

Figure 3.3: 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. We cannot change the altruism by changing the code in our system as altruism is forced by the other party or the peer from whom we are getting the chunks.

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.

Question (Student): Are chunks of larger size stored on nodes with higher bandwidth?

Answer (Instructor): Chunks are of fixed size. But more download can happen from nodes that give better service.

Question (Student): In BitTorrent, can you download and play at the same time?

Answer (Instructor): BitTorrent can be combined with other applications like media player which is downloading and playing. It's better to download enough of the chunk before playing because if the download slows down, the playback will stall.

Question (Student): What does the tracker and Torrent do?

Answer (Instructor): Torrent stores information about the trackers. Trackers keep track of the nodes which have the content. Torrent is a file and we can't statically store information about all nodes that have the chunks because that information can change overtime.

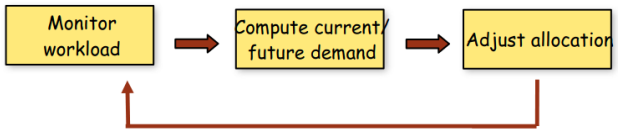
Question (Student): Is BitTorrent a protocol or client?

Answer (Instructor): It a bit of both. It is a protocol as it tells us what the torrent file contains, what does the tracker contain, what is the protocol that we use to connect to the peer.

3.1.4 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 3.4: 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.

3.2 Message-Oriented Communication

3.2.1 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 works better if the processes reside on the same system.
- 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 type of communication works regardless of the processes being on the same machine.
- 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 (Student): Why do distributed systems need low-level communication support for both structured and unstructured communication?

Answer (Instructor): Because distributed systems are spread over a network.

3.2.2 Communication (Network) Protocols

These serve as the established guidelines for communication between processes, facilitating a shared language for inter-process communication. 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.

Question (Student): Can the ordering of the layers change?

Answer (Instructor): The ordering will not change.

Question: (Student): Can the encryption happen at a higher level?

Answer: (Instructor): Only the message can be encrypted, not the other aspects. If enrcypting after Data link layer, nobody can be see what is there in it. However, hardware can be used to encrypt the message itself.

3.2.3 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.

3.2.4 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.

Question (Student) : What is the difference between a socket and a port?

Answer: (Instructor): A port number is the identifier of a socket and a socket is the abstraction.

Question (Student) : How many users can connect to a port?

Answer: (Instructor): Any number of users can connect to a port.

Question (Student) : Every time a client connects, is there a socket connection?

Answer: (Instructor): Yes, every client has its own TCP/IP connection to the server.

Question (Student) : What happens if the same client has multiple requests?

Answer: (Instructor): Depends on how the client is implemented. It can be a sequential client if it's in a while loop sending one request at a time or if the client has multiple threads, they can open multiple TCP connections to the server and send requests in parallel on each of these connections.

Question (Student) : How many processes can listen on the same port number?

Answer: (Instructor): There can be only on socket that is associated with a socket of a certain port number. For example, you can have multiple web servers but these should listen on different port numbers.

Understanding TCP Network Overheads

Normal Operation of TCP: 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. For a single request and single response this is a high overhead but if there is lots of communication between the client and the server, these extra 9 messages wouldn't be a huge deal.

Transactional TCP: There were efforts made to reduce the overhead. Transactional TCP is a protocol that look like TCP. Instead of sending 9 messages as in the above case, here we batch up the messages to reduce the total number of messages going to and fro.

Question (Student) : What happens when there are multiple client requests?

Answer: (Instructor): Typically, it depends on the protocol. In sequential processing, the client sends a request and then waits for an answer. There can be multiple requests with the same connection. You can send a request before the response to the first request is received only in some protocols. This requires tracking which question this response corresponds to.

Question (Student) : What happens if the client doesn't receive the response?

Answer: (Instructor): All the messages have a timeout. If the acknowledgment is not received by a certain time, the request will timeout and the client then has to send it again.

Question (Student) : Is a separate acknowledgement required for each message?

Answer: (Instructor): Packets are sent in sequence, so if acknowledgement for a message is sent, then it means all the messages before it were received, so you can send just one acknowledgement for multiple messages.

Question (Student) : Why does the server send the SYN meeage?

Answer: (Instructor): SYN message is used to establish one-way connection. Throught he first SYN, the client establishes a one-way connection to the server. Only client can send messages to the server. In order to send messages to the client, the server needs to send a SYN message to establish one-way connection to the client. This will then form a duplex connection.

Question (Student) : What happens if some messages in the middle get lost in a sequence of messages?

Answer: (Instructor): Acknowledgement of the first few messages will be sent. If the acknowledgement is not received after the previous ACK, it is assumed that the message after the previous message was lost so the client sends it again.

Question (Student) : When will the server refuse to connect with a client?

Answer: (Instructor): As long as the socket on which it is listening is open, it can continue to receive new connections. The only way to stop is to close the conection.

Question (Student) : What does teardown the connection mean?

Answer: (Instructor): Similar to closing the file, so that we cannot read or write to it.

Question (Student) : Reason why transactional TCP didn't get implemented?

Answer: (Instructor): To implement any change to any network protocol every computer in the world needs to be changed. There is no drawback, it is just hard to deploy in the real world.

3.2.5 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*. Not all machines have multicast capabilities. If we want to do multicast regardless of the underlying hardware we have to do some processing at the application layer. This is basically a library at the application layer that implements multicast as multiple unicast (one to one) messages.

3.3 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: (Student) : If we absolutely need to pass pointers in an RPC, how do we achieve that?

Answer: (Instructor): 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.

Question (Student) : Does the connection stay open from the client side?

Answer: (Instructor): Depends on the stub code. It can decide to setup a new TCP/IP connection for every RPC or send multiple RPCs on the same connection.

Question (Student) : When we write a message from the client, do we need to know the format of the message?

Answer: (Instructor): You don't write a message but invoke a function. Thus, only the function call needs to be constructed with the corect parameters which is then sent as a function call. The rest is taken care of by the stub.

3.3.1 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). This is how complex data structures are sent and restructured in different machines. 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.

Question (Student) : What is the advantage of using RPC over HTTP?

Answer: (Instructor): HTTP is an application-level protocol used for sending and getting replies. But RPCs allow any two applications to communicate with each other.

3.3.2 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.

Question (Student) : What if there are two servers/functions with the same name?

Answer: (Instructor): To avoid collisions we need to make sure that the name is unique, we can just add a version number to the name to make it unique.

Lecture 4:RPC

Lecture 3 cont.

Questions on Failure semantics:

Q: What if the replies from the server were lost?
A: The client did not receive a response here so it will send a request again, this will be a problem when the server is making stateful responses - it will give an error saying the same request is being processed again. Solution is to make all the operations idempotent, irrespective of how many times a request is sent, it will be executed only once and the same response will be sent back every time.

Q: How does a system become idempotent?
A: This is a server side issue, if the server receives same request again it should execute it only once. The client is unaware what is happening at the client side.

Q: For example: Every time a person makes a payment, it generates a new transaction id, in this case how does the server know if its the same request? (here the server is idempotent)
A: The application will re-try with the same rpc request with the same transaction-id.

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. But there is a lot of protocol processing overhead from the packets sent over the network which are returned back as it is the same machine. This can be avoided by optimizing the RPCs. Lightweight RPCs are the special case of RPCs which are optimised to handle cases where the calling process and the called process are on the same machine. They help in reducing the runtime.

Remember the two forms of communication of a distributed system – explicit (passing data) and implicit (sharing memory). You can think of using a special RPC system where both processes are on the same machine but, using a shared piece of memory instead of network messages. 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 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 bit/flag 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,

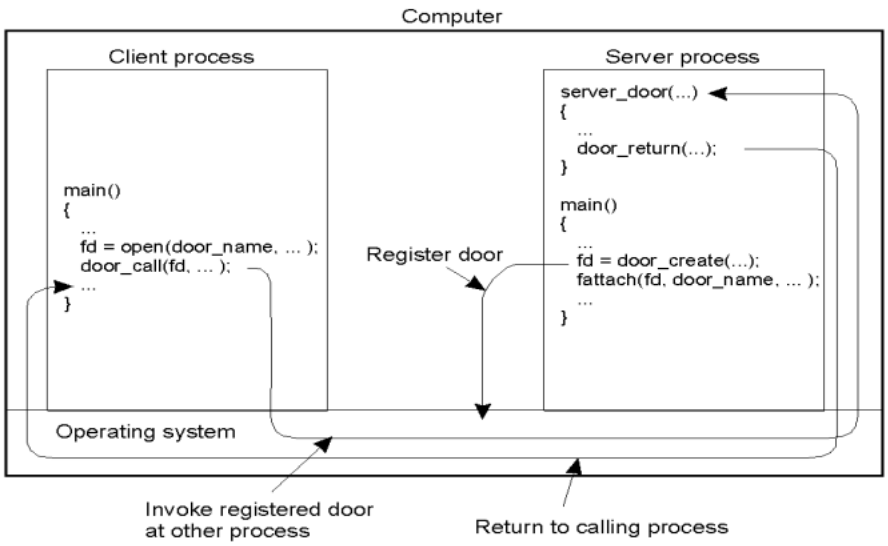


Figure 4.1: Lightweight RPC

Q: Can you use simple IPC instead of LRPC?
A: Yes, we can use IPC when both server and client are on the same machine, but shared memory was the preferred option here.

Q: First step in RPCs is a lookup where we try to locate the server, in LRPCs how do you figure out where the server is?

A: There is an inherent ability in LRPC to figure out where the server is running, it doesn't need to find the network port but it finds the process-id to communicate with the server.

Q: Where is the shared memory stored and is there a capacity limit?
A: Most OS have a concept called shared memory, for example: C/C++ uses malloc/new that allocates memory. In a similar way an OS can make a call which says 'make a shared memory segment' which it can explicit share with other processes. Shared memory segments are specifically designed to be shared across different processes, by default sharing memory is not allowed.

Q. How is the runtime bit determined?
A: It is automatically determined during the compile time or runtime by the program.

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.

Q: Is this unstructured communication where we are using memory regions to pass information back and forth?
A: To some degree it is a hybrid as we are still using RPCs which are structured from our perspective, but the RPC implementation uses unstructured method for a client and server to communicate.

Q: Is this like memory mapped i/o in an internal device?
A: In the memory mapped i/o, hardware device can directly write to a memory region in the OS as opposed to using the CPU path which is somewhat similar but here we take a memory page and share it across two different processes and both of them can read or write.

Q: If there is transparency, how does the client know if it is on the server?
A: In this case it is not transparent otherwise we would not know. we would use the normal path and do standard RPC.

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.

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

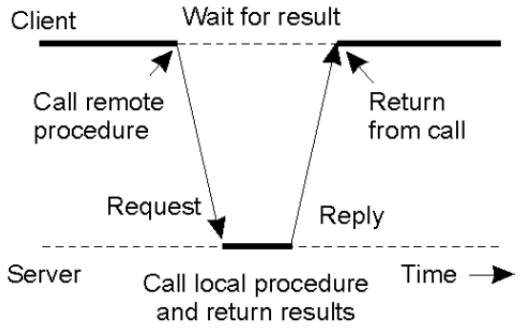


Figure 4.2: Traditional (Synchronous) RPC

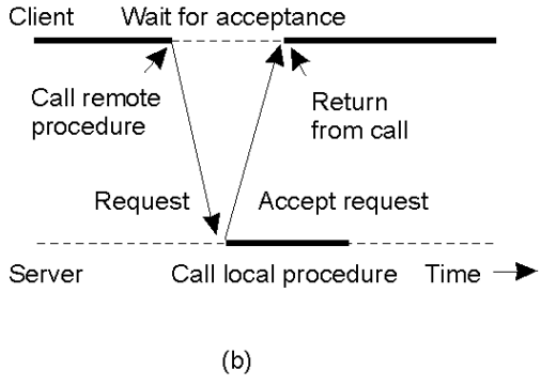


Figure 4.3: Asynchronous RPC

the server responds via its own asynchronous RPC call after completing the processing.

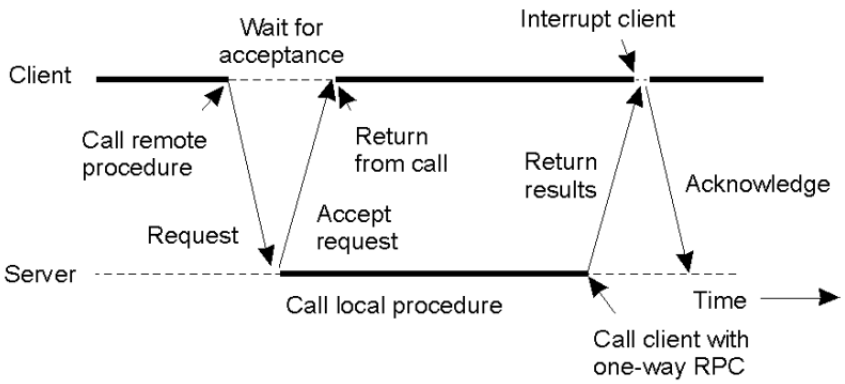


Figure 4.4: Deferred Synchronous RPC

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.

Q: What if there is a time difference between the servers? will we be blocked or not?
A: In this case there is no assumption that the replies have to come back at the same time. Servers will execute at different time and the results will come back at slightly different time. We wait until all the replies are received.

Q: What happens if one of the packets to the servers get lost? A: There are network multi cast protocols that will re-transmit if some server does not receive the packet. So we don't need to handle that at the RPC level. If we don't use the protocols, we need to handle it the RPC level i.e., wait for an act from each server to make sure call was received. If not we have to re-transmit.

Q: How does a normal call become an RPC?
A: When we write the code, we have to declare which functions are local and which functions are remote. The client makes a function call and that comes to the RPC library. If it is a local call, it is in the same code. If it is a remote call, then the server sends the message.

Q: What is the difference between asynchronous RPC and Deferred synchronous RPC?
A: In asynchronous RPC, the response from the server comes back as a call back. In deferred synchronous RPC, the responses comes back as an RPC request.

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. The server defines the interface and implements interface methods.
2. The client looks up a server in the remote object registry and uses normal method call syntax for remote methods.
3. It supports system-wide object references i.e parameters can be passed as object references here (which is not possible in normal RPC)

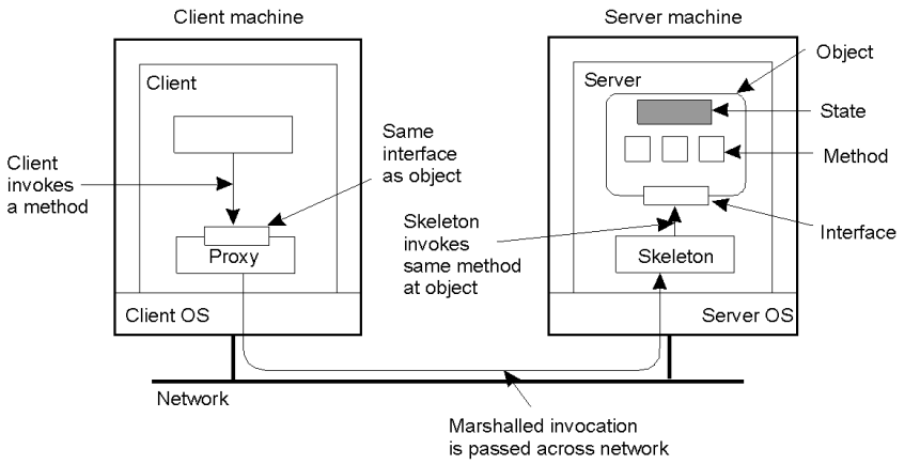


Figure 4.5: Distributed Objects

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.

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.

Q: Can Machine C make a call to O1 by reference? (Look at the figure above)
A: No, because the value of O1 is copied on Machine C. Call by reference only happens when the object is strictly a remote object / UnicastRemoteObject.

Q: What is the difference between RPCs and RMIs?
A: In an RPC you can only make call by value, whereas in an RMI the default call is call by value, but it has an exception where special remote objects can only be called by reference.

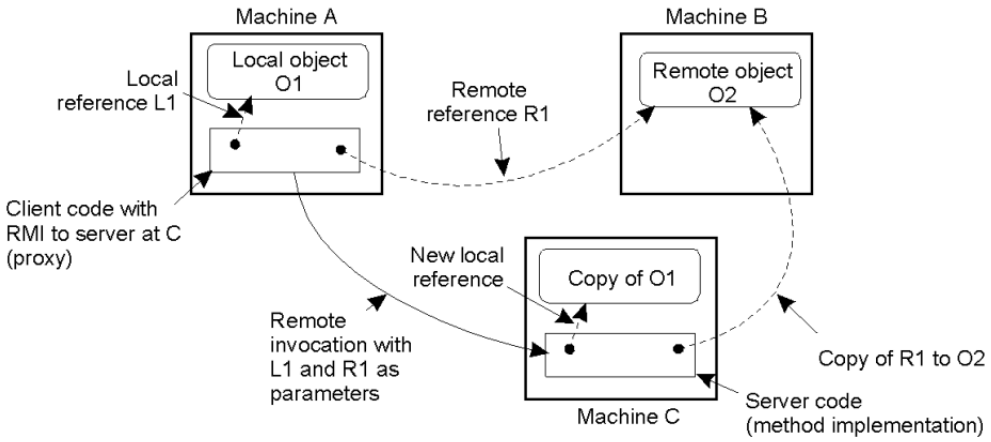


Figure 4.7: Parameter Passing : RMI

Q: How is the memory allocation work for a remote object on a local machine?
A: Since remote objects are passed by reference you don't have the actual object. When you call a method on that object, it is essentially going to go as a message over the network to that object.

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.

Q: If we pass a pointer, will that cause a security issue?
A: There is no security issue because we cannot read some other servers memory. We cannot derefence a pointer on an another machine.

Q: When we pass an object by reference, how will the machine C know how to get to that?
A: When we pass the object we have special type of object called UnicastRemoteObject. These are special types of objects in java. Whatever we pass it by reference contains information of where the object recites which can be dereferenced.

Q: Would machine C connect to machine B and make another call?
A: The programmer would just call O2.foo. The RMI system will then open a connection and send a message to invoke just as you call any other RMI method.

Q: Do RPCs exist because they are faster than HTTP?
A:RPC is preceded HTTP by 30-40 years. RPCs were the standard way to write distributed applications other than socket programming. SO we cannot say if one is faster than HTTP.

Q: Why would you continue to use RPCs instead of HTTPs?
A: HTTPs has web over processing overhead thst not every application neeeds. It is only used if we are using technology.

4.5 Java RMI

- Server:
  - The server defines the interface and implements the interface methods. The server program creates a server object and binds it to the registry called as "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.

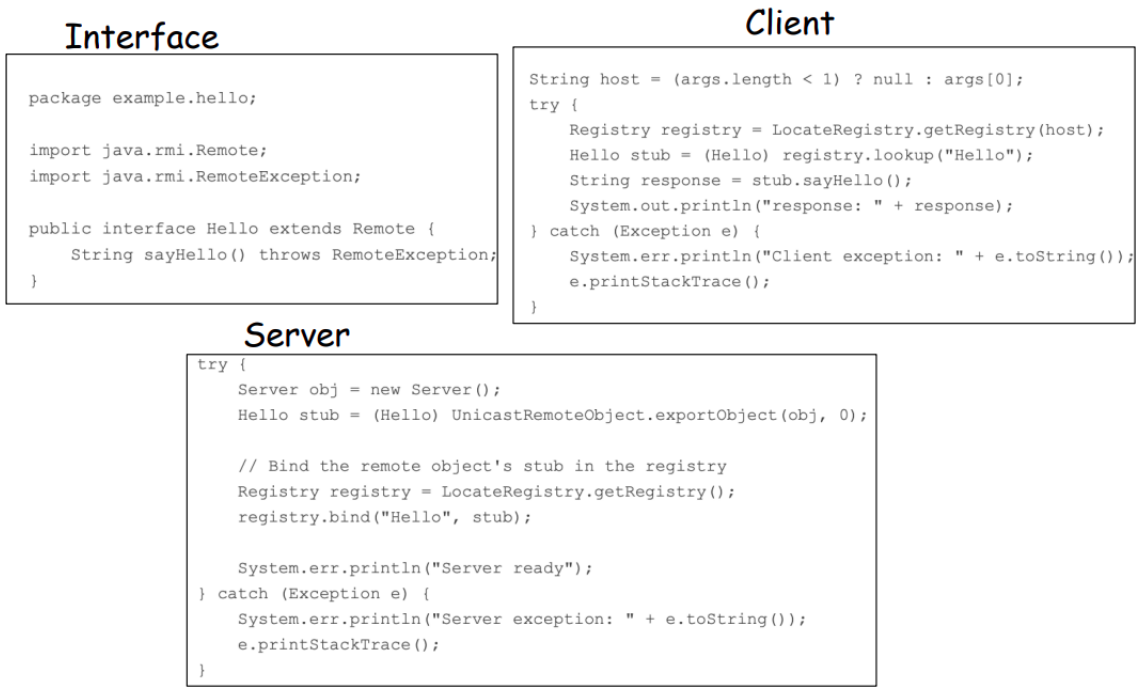


Figure 4.8: Java RMI Example Code Snippet

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). This requires an explicit distributed locking across clients.

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.

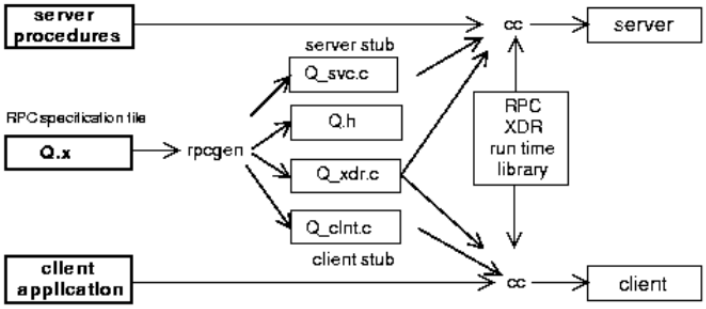


Figure 4.9: rpcgen Compiler

Rpcgen: Generating stubs

There is a **RPC specification file** which lists all the RPC methods that the server exposes and has details like methods names, arguments and more. It is a standard interface description of the server. A special compiler called the **rpcgen** compiles this specification file and automatically generates code called the stub code.

Q: Where does the specification file reside?
A: The specification file has to be common for both the server and the client and has to be created by the development team.

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

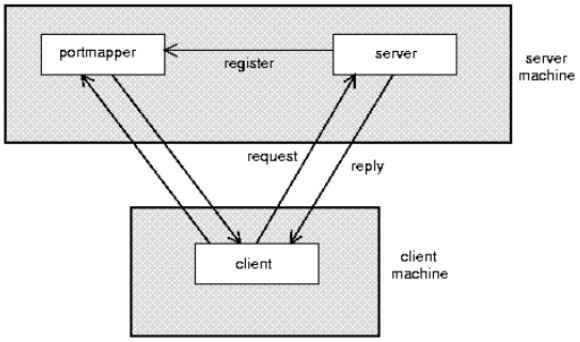


Figure 4.10: Port Mapper

Q: When you register with a port mapper (binder), how do you identify the server?
A: The server has to be given a name, the client needs to know this name and the request sent by the client is executed at the assigned port number.

Q: If the port mapper identifies the port, does the client need to identify where the server is running (ip address)?
A: RPC systems atleast require you to figure what machine the server is running, but not the port itself. There can be two ways to handle this - first, the client can figure out by connecting to the port mapper on the machine. Second, there can be a network wide service lookup where you can search using the server name and get the location of where the server is running.

Q: Does there have to be a mapping of the host name?
A: That is done by the DNS, they can translate machine names to ip addresses.

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 as 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: Is PyRO wrapping RPC?
A: Think of PyRO as a RPC runtime in python, it is going to do all the heavy lifting and simplify implementation for you.

Q: Is PyRO compatible with Python 3?
A: Yes

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.

Q: How do you handle dynamic typing of the language?
A: There are restrictions when we are calling remote objects, sending parameters and so on. There are RMI and RPC systems where we can do dynamic typing.

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.

Q: Is gRPC faster than RPC?
A: It depends on the design because there are http overheads in gRPC. Standard request-response will be

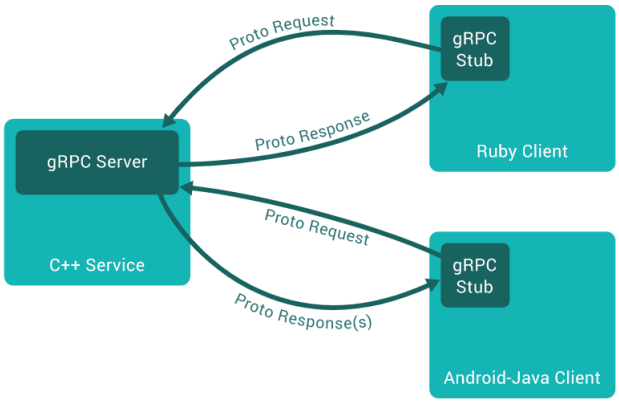


Figure 4.11: gRPC

faster in TCP but gRPC supports 4 types of RPC calls - unary, server streaming, client streaming and bi-directional (see section 4.8.3). A standard RPC is unary (including rmi and pyro) and cannot do any kind of streaming. So gRPC amortizes the overhead by streaming.

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: Why does PyRO not need a ProtoBuf like system?
A: In PyRO, both the client and the server are python programs. PyRO has its own data formats and it has internally declared how messages and arguments are to be sent. ProtoBuf is helpful for language independence which is a feature of gRPC.

Q: Since python language is interpreted do you need to know method definitions before?
A: Yes, because interpreted doesn't mean you can bind at runtime. The client needs to know what the server methods are.

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.

Lecture 5:Threads and Concurrency

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

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 single process runs on a single processor or core at a time.

A **thread** is a light weight 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, own program counter, own copy of registers and control flow as they execute different parts of the process. Threads can run on different cores at the same time. Synchronization might be required if threads are accessing shared data structures and it can be achieved by locks, semaphores, etc...

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 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 run time decides how much stack space to allocate to each thread. The heap is shared across threads.

Threads example:

Single threaded program:

```python
from time import sleep, perf_counter

def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

task()
task()

end_time = perf_counter()
```

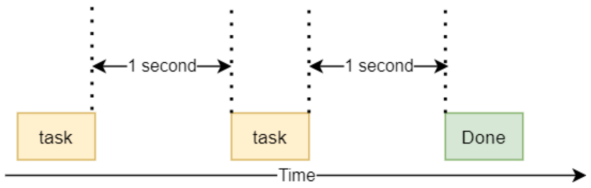


Figure 5.1: Single thread program and its execution timeline

The above python program exhibits basically a sequential execution and there is only one thread. The *task()* function is executed first and it sleeps for 1 second. After 'done' is printed by the *print()* statement, another *task()* function is executed with subsequent sleep.

Multi threaded program:

The above python code shows multi-threaded program where two threads *t1* and *t2* are created by the main thread. Then start the threads by using *start()* function. Depending on the CPU scheduling, when *t1* is in idle state, *t2's task()* function will be executed concurrently. *join()* function enables main() function to wait for the threads to complete.

Question: Does running a code count as its own thread?
Ans: When a program is executed, a main thread is created by interpreter to execute main() function. As per the program in Figure.2, there are two other threads are created by the main thread. So totally there are three threads.

Question: What is the use of join() function at the end in the program (Figure.2)?
Ans: the join() method is used to wait for a thread to complete its execution before moving on to the next operation. the join() method is used to synchronize the execution of multiple threads in a Python program, ensuring that all threads complete their tasks before the program terminates.

Question: What happens if the stack of the thread is full?
Ans: Usually, the stack space is huge. But there are chances for it to be full when there is an infinite for loop or infinite recursion. The thread then throws memory full error and it stops executing.

Question: If the thread terminates due to an error, will the whole process halts too?
Ans: It depends on the type of the error. For example: If there is an illegal memory access, then the whole process will be terminated, whereas for any other normal errors, thread simply exits and the process continues to run.

Question: Is the join() method a blocking call? Will any code after join() be executed after the thread completes execution?
Ans: Yes, it is a blocking call that is waiting for the thread to complete. The main() function will continue and the statements after join() will execute once the thread execution is complete.

Question: Is the above example(Figure 2), is the code running in a single core?
Ans: Yes, it is assumed to run in a single core so parallelism is not observed whereas only concurrency is observed. When there are multiple cores, then two threads can run the program at the same time observing parallelism.

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,

```python
from time import sleep, perf_counter
from threading import Thread


def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

# create two new threads
t1 = Thread(target=task)
t2 = Thread(target=task)

# start the threads
t1.start()
t2.start()

# wait for the threads to complete
t1.join()
t2.join()

end_time = perf_counter()
```

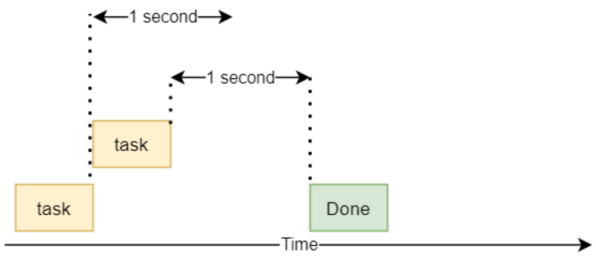


Figure 5.2: Multi thread program and its execution timeline

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.

To make a single-threaded process concurrent, we need to make system calls non-blocking. 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).

Question: What is the difference between each thread sharing address space and yet having its own stack?
Ans: Threads have their own stack and also it can access any address space in the process since it is shared.

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 (dispatcher-worker architecture). 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.

Question: Is there a limit on no. of threads in a thread pool? **Ans**: There is no real limit on the no of threads, the only real limit is the memory.

5.2.2 Concurrency Models

The type of threading system we use in the server becomes our Concurrency Model. All concurrency models use the same abstract model - Server is an infinite loop waiting for requests. 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.

Sequential Server

It is a single threaded server that processes incoming requests sequentially. The server will wait for the request and process it one by one.

```
while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```

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.

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. Lifetime of a thread is the lifetime of the request, which is short-span.

```
while(1){
  req = waitForRequest();// get next request in queue
                         // wait until one arrives
  thread = createThread(); // start a new thread
  thread.process(req); // assign request to thread
}
```

  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.

```
CreateThreadPool(N);
while(1){
    req = waitForRequest();
    thread = getIdleThreadfromPool();
    thread.process(req)
  }
```

  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.

  Question: How do we know the thread is idle?
  Ans: The idle thread will be in the thread pool. The mechanism will be done while creating the program.

  Question: If there are N threads, will main thread included in the N number of threads?
  Ans: No, main thread will spawn N threads, so the program will have N+1 number of threads.

  Question: Is N configurable? **Ans**: N is only cofigurable at the server startup time. Once the server starts, we can't change N. We need to terminate the server, change N and start the server again.

- **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 low threshold, we increase N and start new threads and add it to the pool.
  – If the number of idle threads is greater than high 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

```python
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())
```

```python
def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()
```

Figure 5.3: Asynchronous and synchronous version

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.

Note: If you make a function async, you need to make all blocking calls inside the function also async. Refer `https://python.plainenglish.io/build-your-own-event-loop-from-scratch-in-python-da77ef1e3c39` for more details.

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.

Question: What happens after the sleep of first count() in Figure 3?
Ans: It has to wait for whatever it is being executed to be completed and yield control.

Question: Who is monitoring whether the function is over or not?
Ans: The asyncio library is responsible for monitoring the status of functions (i.e., asynchronous functions) and determining whether they have been completed or not. It is managed by the programmer.

Question: Where is the speedup compared to dynamic thread pool?
Ans: There is only one thread involved in this model but it behaves as a multi threaded program. This means that there is no OS scheduler involved in switching threads which may lead to increase the speed. It is not to be used as a way to speed up but as a way to achieve concurrency in a single threaded program.

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. For example, Apache web server uses a process pool model which also supports multi threading.
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.

Summary

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

- Pure sequential
- Event based
- Thread based
- Process based

Which architecture is more efficient?
The answer depends on what kind of machine it is being used on.
For single core machines- event-based model is more efficient than thread-based as there are no context switching involved.
For multi core machines - thread-based server yield high performance because they can achieve true parallelism.
For best performance, we need a multi-threaded event based architecture.

5.2.3 Parallelism vs Concurrency

- **Concurrency** enables us to handle multiple requests.
  - Request processing does not block other requests.
  - This can be achieved using threads or async (non-blocking) calls.
  - Concurrency can be achieved on a single core/processor.
- **Parallelism** enables simultaneous processing of requests.
  - Request processing does not block other requests. Requests are processed in parallel.
  - This can be achieved using multiples threads or processes as threads or processes can run on multiple cores/processors
  - Need the right hardware support. Multiple cores/processors are needed.

5.2.4 Part 3: Thread Scheduling

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. Two-level scheduling happens. The OS picks the process, the library scheduler picks the thread to execute.

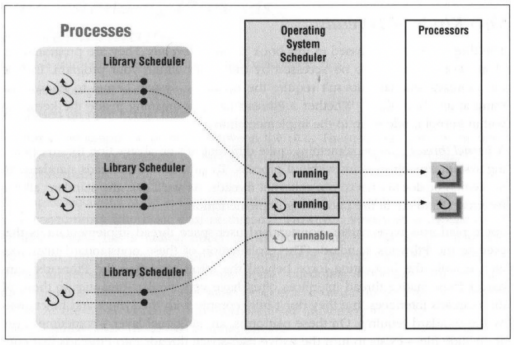


Figure 6-1: User-space thread implementations

Advantage:
Creation of threads is very lightweight because it doesn't require system calls. It offers flexibility to the programmers as they can fix the scheduling algorithm.
Disadvantage:
Threads compete with each other. They 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.

Kernel-Level Threads

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

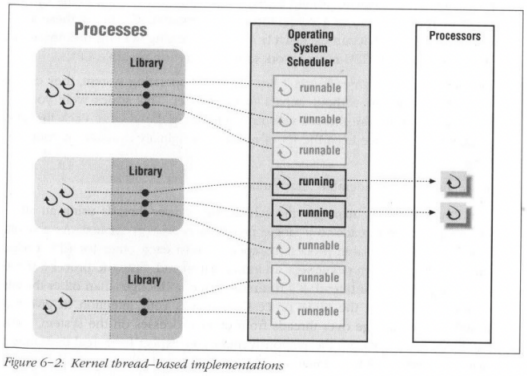


Figure 6-2: Kernel thread-based implementations

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. Creation and deletion of threads is more expensive as it results in a system call to OS.

Question: Is there a particular algorithm that OS uses to pick threads ?
Ans: OS scheduler algorithms are used to pick threads. It is much similar to process scheduler.

Scheduler Activation

- User-level threads use two-level scheduling: OS scheduler and Library scheduler. The kernel might switch user-level thread during an important task. So information passing between kernel and library is needed.
- **Scheduler Activation** is an OS mechanism for user-level threads.
  - Kernel notifies user-level threads about kernel events with calls like I/O is done, CPU available etc...
  - Library informs the kernel to create/delete threads. N:M mapping - N user-level threads mapped onto M kernel entities.

Lightweight Processes

- Lightweight processes sit between threads and processes. Instead of creating threads per process, several LWPs are created per heavyweight process amd threads are mapped onto these LWPs.
- Each LWP when scheduled searches for a runnable thread (two-level scheduling).
- When a LWP thread is blocked on a system call, OS context switches to another LWP.

Lecture 6:Computing Demand

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:

Multiprocessor scheduling
Distributed scheduling
Case Studies : V-System, Sprite, Volunteer Computing, Condor
Cluster Scheduling

6.2 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 present at each of the processors which will speed up the performance of anything that is executing on those 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 dedicated to processors. Memory, or RAM, are shared across all of the processors using system bus (represented by the blue line in the diagram); a program running on one processor can access any address in memory. Multi-processor scheduling involves scheduling tasks in such an environment.

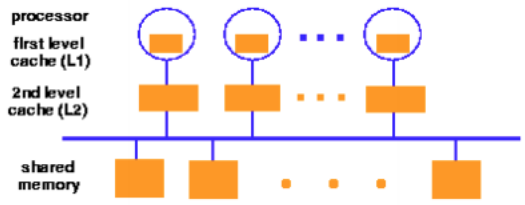


Figure 6.1: Multiprocessor scheduling.

6.2.1 Central queue implementation

In a central queue, all of the processors share a single global queue or run queue where all the threads and processes are present. Execution of a process happens one quantum at a time and 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. This is essentially what happens in a uni-processor system.

6.2.2 Distributed queue implementation

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

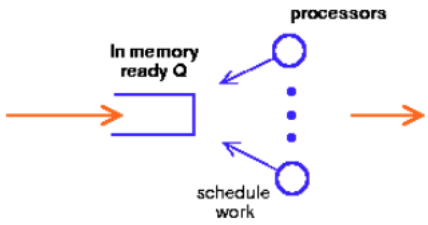


Figure 6.2: A central queue.

take the next job in that queue to schedule for execution one quantum at a time.

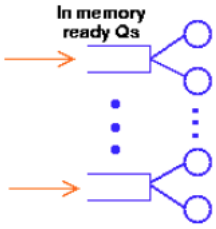


Figure 6.3: A distributed queue.

6.2.3 Pros and cons of the 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 time-slices which will be wasted.

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 CPU's time than others. Tasks in shorter queues get more round-robin timeshare. To deal with this, every once in a while we need to re-balance this queue if there is imbalance in the load by looking 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 instructions for that job stored in the cache. The process goes to the end of the queue, eventually appears at the front, and gets scheduled at another processor. There is no mapping of tasks to processors in the centralized queue. That processor will start from a cold cache with no data and instructions for this job. The initial instructions will all be cache misses. Program execution is slowed down to warm up the cache, and then the time slice ends. That is why the central queue has poor cache affinity.

This argues for using distributed queue-based scheduling. Once a 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 a warm cache. Once in a while, the process could be moved to another process for load balancing where it will start from a cold cache, but it is not a common scenario. Even though you might try to schedule a 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 miss. 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 a CPU scheduler, Cache Affinity plays the most important part. It should be considered over synchronization bottleneck and lock contention. So, there are a 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.

6.2.4 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 will increase the waiting time.

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 this issue too, by adjusting the priority. This will hold off from relinquishing the CPU allowing the other threads to continue to do useful work.

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.

Q: What is spin-lock?

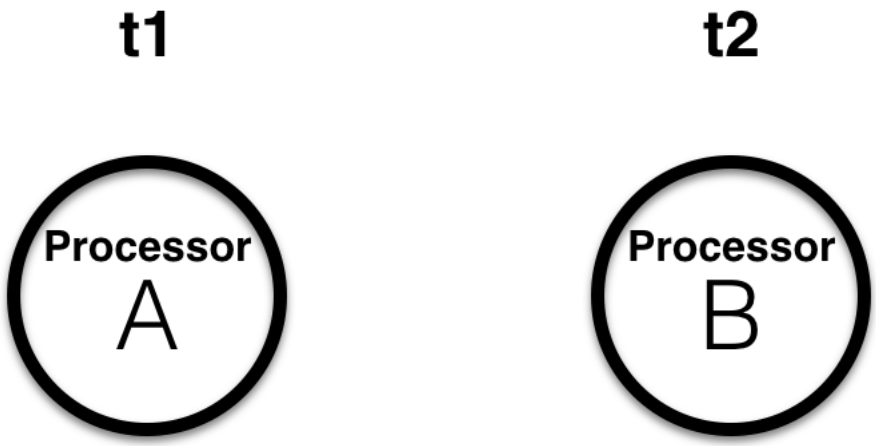


Figure 6.4: Two CPUs, and a process with two threads.

A: There are many ways to implement a lock. One form is while the thread is waiting for the lock to get released, it will wait in a loop ("spin) while repeatedly checking if the lock is available. It wastes cycle but it is one way to implement a lock. It is a form of busy waiting. In some parallel application that's how you implement a lock. Other way is if the lock is not free, the thread blocks the process.

Q: If a machine has four cores and a process runs on two cores, what will happen to the other two?

A: That depends on the scheduling policy of the machine. If it implements gang scheduling, another process that requires two threads and schedules them. If it is general purpose scheduler, you can schedule any other process in the queue.

Q: How do you handle gang scheduling when there are more threads than processors?

A: In gang scheduling, if you have more threads than processors, for instance 6 threads but only 4 processors, you schedule 4 threads to run simultaneously on the 4 processors. The remaining 2 threads will need to wait until a processor is available. This approach might require managing which threads were previously run to maintain efficient scheduling. Ideally, the number of threads should match the number of processors to avoid this complexity. For example, with 4 processors, optimally you would have 4 threads, ensuring each processor is dedicated to a single thread, simplifying the scheduling process.

6.3 Distributed scheduling

6.3.1 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 executes the job locally. In distributed scheduling, you have an additional degree of flexibility. Even though the user submits a job at machine I, the system may actually decide to execute the job at machine g and bring back the results. The basic reason is the harvest idle cycles on workstations. Referred to as scheduling in a Network of Workstations (NoW). Distributed scheduling - taking jobs that are arriving at one machine and running them somewhere else in the system - does this make sense (any advantage to this)?

Scenario 1: Lightly loaded system

You should run the 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.

What is 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.

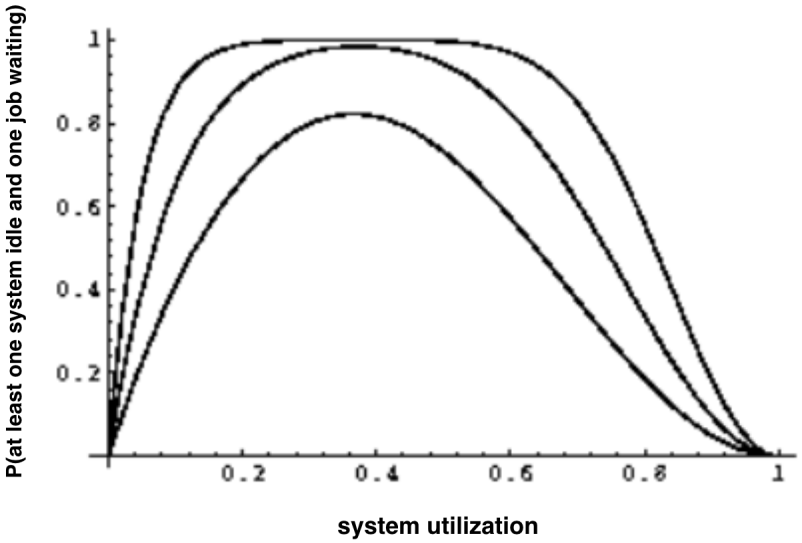


Figure 6.5: The relationship between system utilization and resource under utilization.

- 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.

6.3.2 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 the 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 the job has started executing, the scheduler can still preempt it and move it somewhere else. Transfer of a partially executed task is expensive due to the collection of the 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 the 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 the arrival rate reaches system capacity, machines don't want incoming jobs and send jobs off to other machines that also don'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.

6.3.3 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 has 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 that the overhead from task transfer is compensated by a reduction in the 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 the above information of other nodes should be collected - demand-driven, periodic, or state-change-driven - so that the scheduler can make all of its decisions in the right way.

6.3.4 Sender-initiated distributed scheduler policy

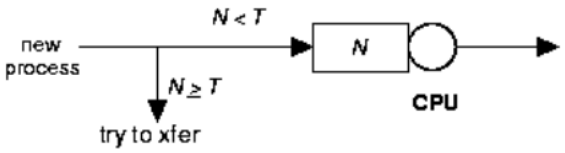


Figure 6.6: A sender-initiated distributed scheduler policy.

The overloaded node attempts to send tasks to the 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 the poll limit has been reached. Transfer the task to the first node below a threshold. If no receiver is found, then the sender keeps the task.
  - Shortest: poll N_P nodes in parallel and choose the 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

Q: Is the location implemented at a machine level where each machine makes a local decision or can it be implemented globally at a coordinator node or a load balancer which keeps track of every machines' load information and makes decision?

A: Both policies exists. What is shown here is a local machine that is querying but the other policy also exists.

6.3.5 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.

Q: Is a receiver-initiated policy redundant or wasteful when it seeks out work?

A: No, a receiver-initiated policy is not inherently more redundant or wasteful compared to a sender-initiated policy. Both policies serve as duals to each other, meaning they essentially perform the same function but from opposite starting points. In situations where some machines are overloaded, they can act as senders and attempt to distribute work. Conversely, machines with lighter loads can act as receivers, actively seeking out work. The effectiveness of either policy hinges on the presence of the counterpart (a sender for every receiver and vice versa). The choice between a sender-initiated or receiver-initiated approach typically does not impact the overall efficiency significantly, but rather influences the time taken to establish a sender-receiver pairing.

6.3.6 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 low load, node acts as receiver. If load is above high threshold, node acts as sender.

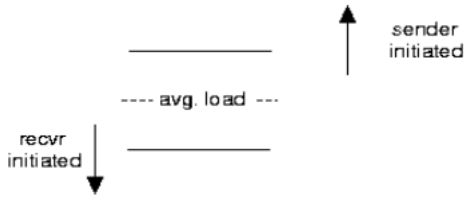


Figure 6.7: A symmetric policy.

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.

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.

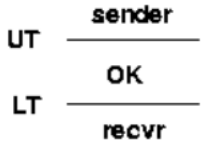


Figure 6.8: An improved symmetric policy.

6.4 Case Study

6.4.1 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. V-system doesn't implement a symmetrically initiated distributed scheduling policy. The machine could either be a sender or a receiver. The middle ground doesn't exist.

Q: Is M constant? Does it change?

A: M is decided based on number of nodes in the environment. M is proportional to the size of the network. Optimum value is picked by the system.

6.4.2 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. This will not be possible if a process is doing network communication because the IP address is tied to the machine and you can't move the process in the middle to another machine because the new machine will have a different IP address and all the socket connections will be broken.

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.

Q: Do processes declare whether they can be migrated?

A: On some degree it depends on what the user wants to do. If the user knows if the process does not require network communication, then yes it can be declared.

Q: Does process migration between two machines require same OS?

A: For Sprite, yes.

Q: Is it possible to move a process to a different hardware setup?

A: Yes, but the hardware must be of the same type. Moving a process across different hardware architectures, like from Intel to ARM, is not feasible due to differences in registers, architecture, and compiled instructions. Even with identical hardware, issues can arise, such as missing files the process was accessing or broken network connections due to changes in IP addresses. However, for processes without dependencies on external files or network connections, simply doing computational tasks, transferring the process can be possible, similar to what was achieved with Sprite in process management.

6.4.3 Condor (U. of Wisonsin)

The distributed system that still exists today. 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. If Condor runs on OS that does not support process migration, then we need to configure condor to not terminate the process when user comes back or terminate the job. 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.

6.4.4 Volunteer Computing

It is a distributed scheduling on a very large scale over WAN. 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.

Q: Is latency a factor about which machines are near and which machines are far?

A: The latency is not as big a problem because let's say you do 30 second or one minute computation typical latency anywhere on the network is going to be 100 some millisecond so you can pay hundreds of millisecond cost and run for of 10 seconds.

6.5 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.

6.5.1 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.

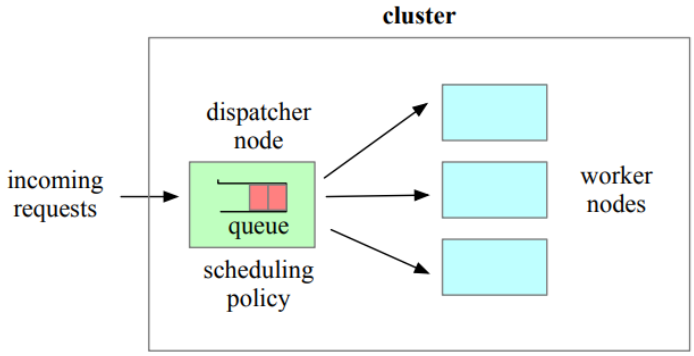


Figure 6.9: A typical cluster scheduler

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

6.5.2 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 the case of stateful applications, for, e.g. an online shopping store, the state of the shopping cart is maintained by the machine which serves the user's 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 the 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 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 a 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 databases 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]

6.5.3 Scheduling Batch Applications

This is used for larger jobs like ML training, data processing, and 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 machines is **SLURM** (Simple Linux Utility for Resource Management). Fundamentally it has a queue and it has a scheduling policy. But it has many other features:

- 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 the runtime of the job. For, e.g. a queue for short jobs would terminate the job if it exceeds the time limit.

6.5.4 Mesos Scheduler

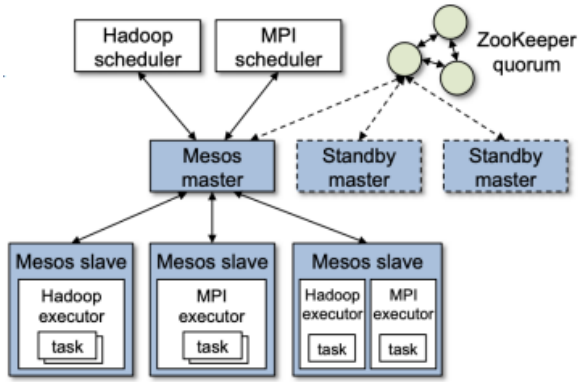


Figure 6.10: Mesos Architecture

The Mesos Scheduler was designed in Berkley but has now 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 the resource needs of different frameworks. It has a hierarchical 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

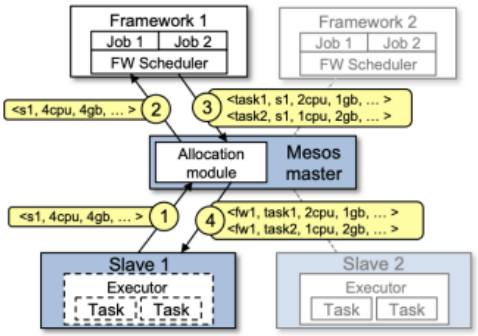


Figure 6.11: Mesos Scheduling

are policies to decide which framework has to offer the resources. Once the task run by the framework is done and the resources are idle, it is available to be re-offered by Mesos. There are 4 components to the scheduler: The coordinator, worker, framework scheduler, and executer.

Question: Does the coordinator offer fractions of machines and not whole machines?
Answer: Correct. That's why it's called a bundle of resources. They don't have to full servers. They can be slices or a collection of slices.

Question: When does Mesos decide to make offers again after rejection?
Answer: That's left to Mesos as a policy decision. No right answer. Every minute, every 10 minutes. It's ok, though, because these are batch jobs, not interactive.

Question: How do you know what size offers to make? **Answer:** No knowledge, but there are some typical configurations. Policy decision.

Question: If there is a busy framework and a not-busy framework, do you want to offer the not-busy framework more resources?
Answer: The framework 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 frameworks.

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 the 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 will eventually get resources assigned.

6.5.5 Borg Scheduler

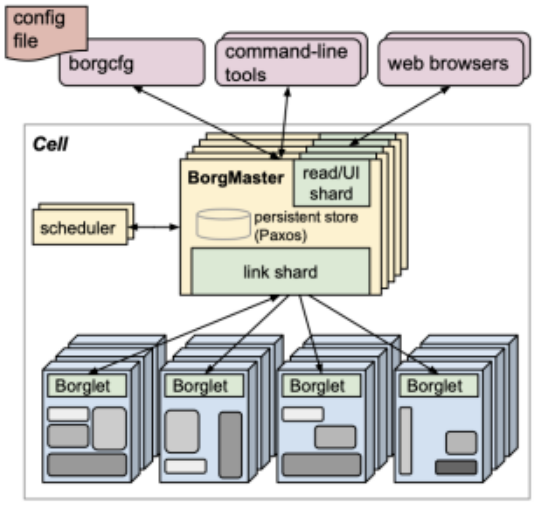


Figure 6.12: Borg Scheduler

This is Google's production 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 a mixed scheduler designed to run both interactive and batch jobs. Interactive jobs like web search and mail are user-facing production jobs that are given higher priority. Batch jobs like data analysis are non-production jobs that 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 the 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 them.

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

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

Question: After termination, does Borg save the work in some memory? **Answer:** Typically Borg is not responsible for saving state, just resources. Developers might add checkpoints if their job is low low-priority

Question: What's the point of a cell? **Answer:** Let's take a look at multi-tiered applications. We need 3 machines for the Interface, Application, and Databse. Or a clustered webserver. Of course you can ask for just 1 machine.

Question: Are cells created proactivly or reactivly **Answer:** Generally "best fit" on set of machines. You don't have to take the whole machine. Large services take a group of machines

Question: Can there be a single point of failure for interactive jobs. **Answer:** If the scheduler goes down, the whole system goes down, and if the application fails, there is 5* replication to avoid this. Generally seperated geographically at different data centers. The Borg monitors applications and will restart cells or machines in the event of application failures. No state saves, however.

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 reclaim the servers. If the job being terminated is a web server, it doesn't make sense to pause it. But if it 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 a 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.

Lecture 7:Thread Scheduling

7.1 Overview

This lecture covers the following topics:

Part 1: Computing Demand Introduction

Part 2: Computing Demand Sustainability

Part 3: Using Computing To Improve Broader Sustainability

7.2 Lecture Notes

7.2.1 Part 1: Computing Demand Introduction

Computing Demands Increasing And Implications

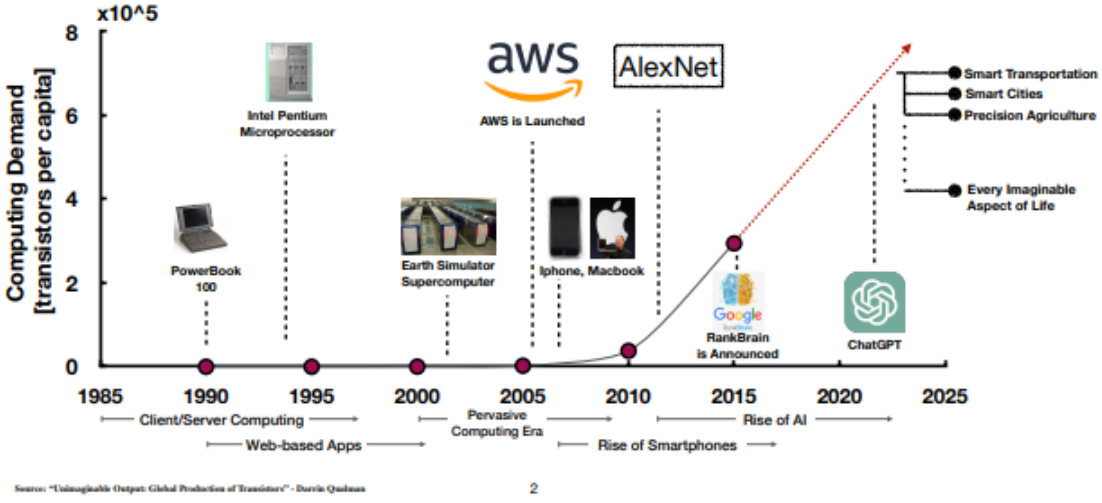


Figure 7.1: Computing demand vs time

As shown in figure 6.1, the key inventions of earlier years were not as frequent and didn't greatly increase demand. Currently, the computing demand is increasing exponentially and will continue to accelerate in growth.

The following key events have contributed to this increase in demand

1. Introduction of cloud services
2. Smart phone revolution
3. Rise of AI

While energy demands of computing is increasing, computing can be used to lower energy demands in other applications. For example, ride sharing could reduce the energy need to make cars since more people may not find the need to purchase them.

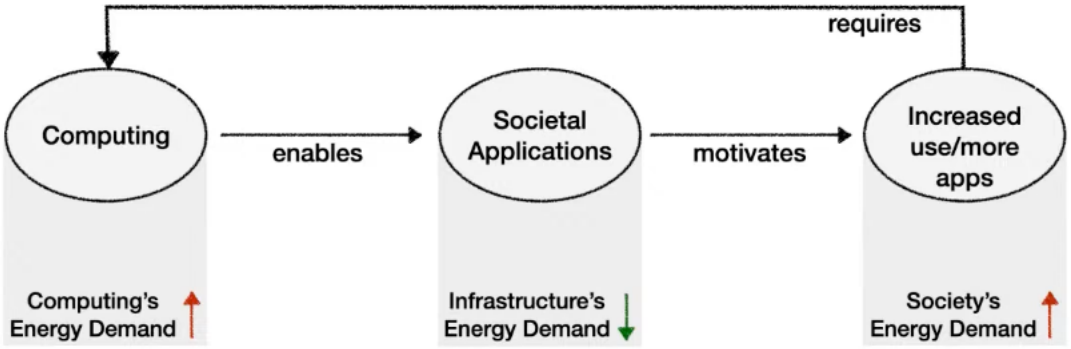


Figure 7.2: Implications of increasing computing demand

Successful apps inspire more people to use it. Jevons Paradox states that when you make an application more efficient, the energy to run the application reduces but the overall usage of the application increases. Thus, society's energy demand for a successful app increases.

Question: What is the effect of time efficiency on energy demand?
Answer: The figure doesn't account for increased energy effects as a result of the time efficiency.

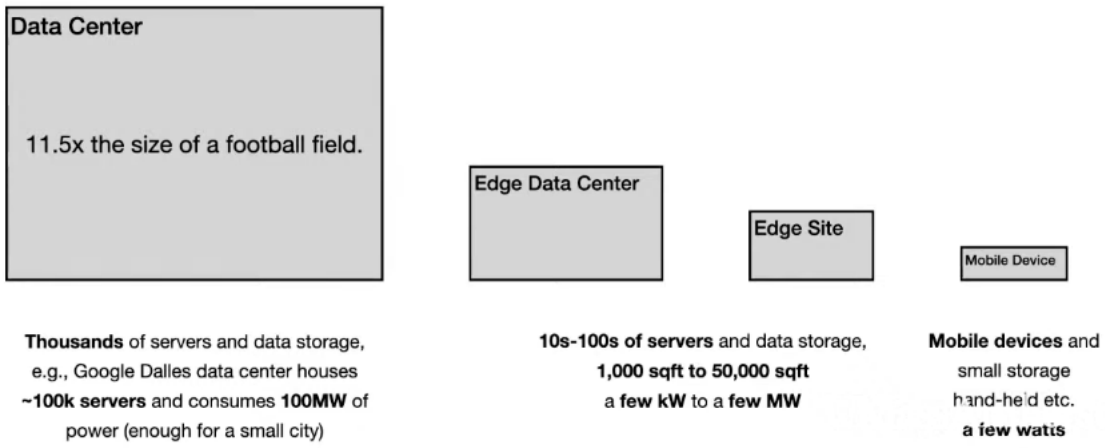


Figure 7.3: Implications of increasing computing demand

How Is Demand Served?

Computing demand can be satisfied as follows (see **Figure 6.3**)

1. Big data centers, which have thousands of servers in them. These are huge facilitates and consume a huge amount of energy (hundreds of megawatts).

2. Edge data centers. These are smaller and have hundreds of servers. They consumes a few kilowatts
3. Mobile devices, which consume few watts of power

7.2.2 Part 2: Computing Demand Sustainability

Contributions To Data Center Cost, Energy And Carbon Footprint

The following describe the cost, energy consumption and carbon footprint of data centers.

Cost:

1. Use of building and location of the data center
2. Replacing servers
3. Energy of data center

Energy:

1. Computing
2. Cooling equipment

Carbon Footprint:

1. **Embodied:** Carbon emissions from manufacturing computing hardware and data center building
2. **Operational:** Carbon emissions from energy used for computing and cooling

How To Serve Demand In A Sustainable Manner?

In the context of computing, **sustainability** is defined as operating the infrastructure in the least carbon intensive way. In other words, to be more sustainable is to emit less carbon when running your servers. Thus, the goal is to reduce embodied and operational carbon emissions. Carbon footprint can be computed as follows

$$\text{Carbon Footprint} = \frac{\text{Cycles per Unit Work} * \text{ Total Units of Work}}{\text{Computing's Energy Efficency} * \text{ Energy's Carbon Efficiency}}$$

Where **Cycles per unit work** is the amount of CPU cycles to perform a unit of work, **Total unit of work** is total work units computed, **Computing's energy efficiency** is how many cycles can be ran on a server per unit energy of work and **Energy's carbon efficiency** is how much unit energy is produced per 1 gram of carbon dioxide.

Motivations In Data Center Innovations

Power Usage Effectiveness (PUE) is a value multiplied by monthly server cost to get the total cost including cooling. In traditional data centers the PUE value is usually 2. State of the art data centers (hyperscale) can have a PUE value of 1.1. There has been a shift to hyperscale cloud providers over the years as shown in figure 6.2, which is caused by the cost of energy.

As shown in figure 6.3, predictions were made that energy demand will double every year, but some estimates show that is not the case, while others think that it has been worse. However, what is clear is that demand will eventually grow as it will no longer be offset with PUE savings.

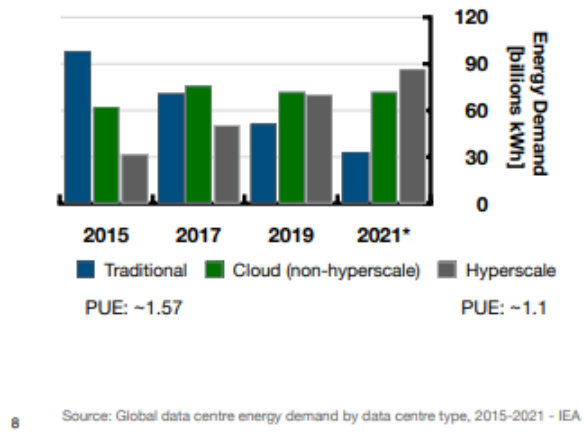


Figure 7.4: Energy demand with data center distinction

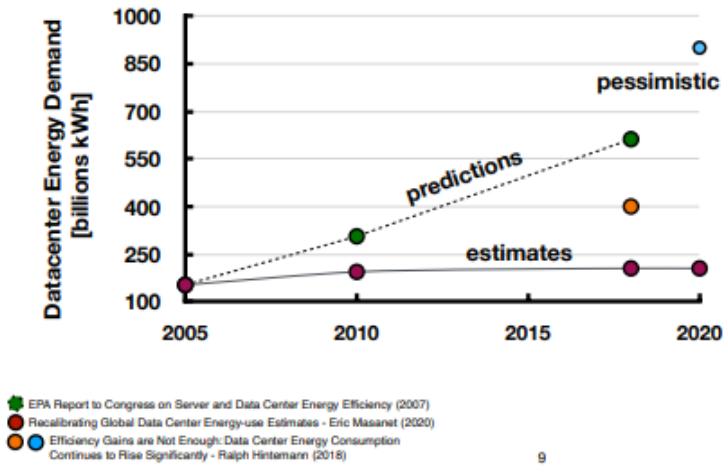


Figure 7.5: Data center energy demand vs time

Opportunities To Improve Carbon Foot Print

The following parameters to the carbon footprint equation are analyzed for their potential carbon emissions savings below

Cycles per unit work: Algorithmic efficiency can help the CPU run work in less cycles, but there are limits. This parameter eventually has a bound.

Total Units of work: This is unbounded, there is no end to the amount of work a data center does. In fact, there is a monetary incentive to increase total work.

Computing's energy efficiency: Laundar's principle states that theoretical efficiency limit of CMOS will be hit by 2050, but could be practically sooner. Kommey's law states that energy efficiency doubles

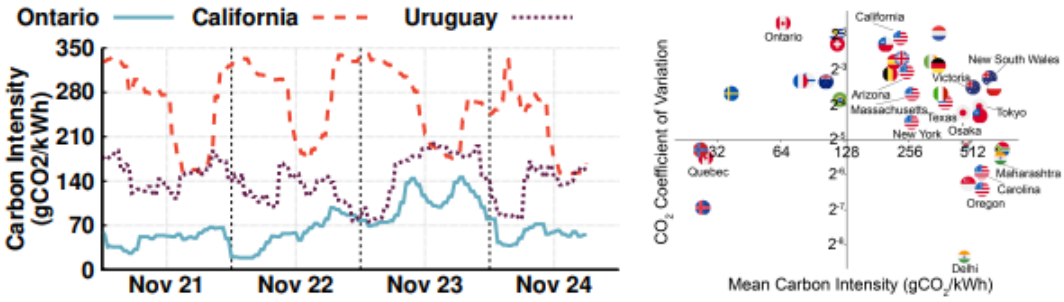


Figure 7.6: Carbon intensity in different regions

every 1.6-2.6 years. Jevan paradox states that gains in efficiency does not reduce demand. This parameter also has a bound.

Energy's carbon efficiency: A lot of gains can be made by transitioning to low carbon energy. Most of the work is done to optimize this parameter.

Grids Carbon Intensity And Clean Energy

Carbon intensity is how much carbon is emitted for a unit of energy created. While some countries have been improving in this metric, it is not possible to control how much carbon intensity the world is emitting.

Since carbon intensity tends to be variable, as shown in figure 6.4, there are opportunities of doing computation when carbon intensity is low. The variation in intensity is due to availability of solar energy. Computing also gives the ability to shift demands into regions with low intensity variations.

Computing Advantages

There are many options for running a processing job

1. Run immediately
2. Run somewhere else
3. Run later
4. Run slower/faster
5. Run intermittently

These options are useful when it comes to carbon intensity. For example, you can run a job faster when intensity is low or move a job to an area with less carbon intensity.

Why would companies want to improve sustainability? One reason is that customers care about it and the company wants to look green in their eyes. Additionally, companies selling other sustainable products need to know about their computing carbon foot print.

To read more about "How can we leverage carbon intensity variations and computing's flexibility?", have a look at the papers on "Enabling Sustainable Clouds: The Case for Virtualizing the Energy System" by Noman Bashir published at SoCC'21, ASPLOS'23.

Accounting For And Reducing Embodied Carbon

It is important to measure the emissions of a computing data center. There is disagreement about whether embodied or operational is more important to optimize. For example 80% of emissions of iphone is embodied, but the opposite could be true for servers. One view is that embodied emissions is for the producer of the good to optimize. In other words, everyone should focus on lowering their operational emission, which in turn lowers both types of emission. Another view is weighting the two equally.

Implications for Sustainable Computing

The following steps should be taken as an action plan for a carbon free grid. First, terms such as "Carbon-free", "carbon-neutral", "zero-carbon" and "100% renewable" should be clarified to prevent false impressions. Next, enhancing carbon visibility is important to know how much carbon you are emitting. Then, the focus should be shifted on carbon and not energy consumption. Finally, using computing flexibility to balance the grid.

7.2.3 Part 3: Using Computing To Improve Broader Sustainability

Computing Use Cases

There are ways to use computing in order to improve the energy consumption of other applications. For example, a building could have sensors to monitor temperature. That data could be analyzed to learn when people are in the building. This can be extended to automatically control lights and HVAC for the building. This is an example of a **Sense, Analyze, Control model**.

Another example is monitoring pests/disease in agriculture or sensing other cars to avoid congestion in transportation.

Power Monitoring And Analysis Of Data

At the home, there is outlet-level monitoring and meter-level monitoring. Data collected from these monitors can be used with machine learning or statistics to figure out patterns and inefficiencies.
Example: occupancy sensor feeds data to a smart thermostat to design schedule for heating a home.

Low Carbon Energy

There has been huge growth in renewable energy. The problem is that the energy is intermittent. One solution to this problem is to use past and weather data to forecast solar energy. A use case for this is electric vehicle charging. By analyzing when solar is high, you can delay their charging until then.

Summary

1. Sustainable Computing
   (a) Demand is growing
   (b) Need to make server demand sustainable
   (c) Use computing unique advantages to optimize carbon footprint
   (d) Reduce operational carbon emission as well as embodied
2. Computing for Sustainability
   (a) Leverage computing so other sectors can reduce energy consumption + enhance use of carbon energy

Question: Does companies claimed PUE correct? How can companies claim to be zero carbon if they are running in places such as india?
Answer: PUE values are accurate. Companies do accounting magic by buying carbon offsets and investing in renewable and gaining carbon credits.
Question: What is your main research focus?
Answer: Lecturer has worked on all angles discussed in this presentation. Leveraging computing to make grid more efficient. Recently, has worked on sustainable computing itself.

Lecture 8:Cluster Scheduling

8.1 Overview

In this lecture, we continued Cluster Scheduling i.e. Borg Scheduler and started a new topic, Virtualization.

8.2 Cluster Scheduling

8.2.5 Borg Scheduler

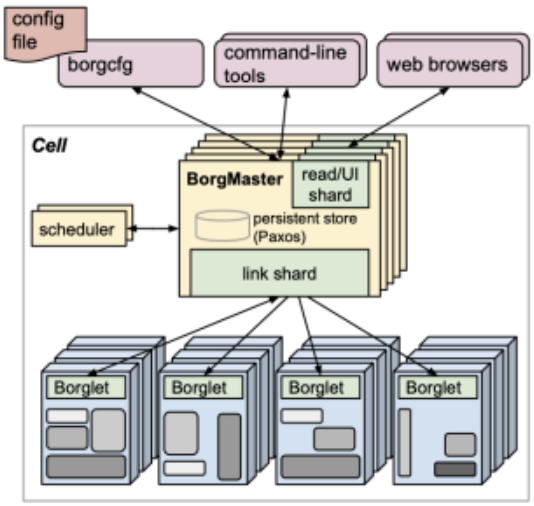


Figure 8.1: Borg Scheduler

This is Google's production 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 a mixed scheduler 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: After termination does Borg save the work in some memory. **Answer:** Typically Borg is not responsible for saving state, just resources. Developers might add checkpoint if their job is low priority

Question: Whats the point of a cell. **Answer:** Lets take a look at multi tiered applications. We need 3 machines for the Interface, Application, and Databse. Or a clustered webserver. Of course you can ask for just 1 machine.

Question: Are cells created proactivly or reactivly **Answer:** Generally "best fit" on set of machines. Don't have to take whole machine. Large services take a group of machines

Question: Can there be a single point of failure for interactive jobs. **Answer:** If the scheduler goes down the whole system goes down and if the application fails, there is 5* replication to avoid this. Generally seperated geographically at different data centers. The borg monitor applications and will restart cells or machines in the event of application failures. No state saves however.

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.

8.3 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: Is there a difference between virtualization and emulation
Answer: Emulation is a type of virtualization in some cases.

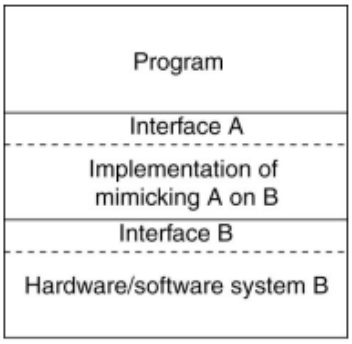


Figure 8.2: Virtualization

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.

8.4 Type of Interfaces

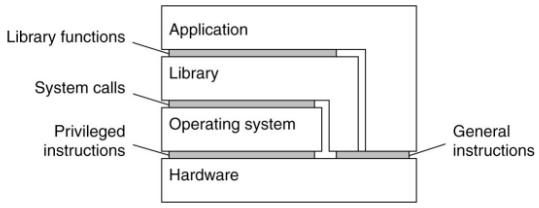


Figure 8.3: Types of interfaces in virtualization

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

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

8.5 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.

8.5.1 Types of Virtualization

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)

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. Since there is no translation of instructions it is much faster than emulation.

Applications of nature virtualization include: Virtualbox, Parallels **Examples:** IBM VM family, VMWare Workstation, Parallels, VirtualBox.

Question: What is difference between Emulation and Full Virtualization
Answer: Emulation is often different from the underlying interface. Think ARM to x86. In full virtualization the interfaces are the same.

- 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.

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.

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.

Question: What is the difference between OS and Hardware level virtualization.
Answer: We have some interface A used to mimic interface B. Hardware is emulating the actual chip.

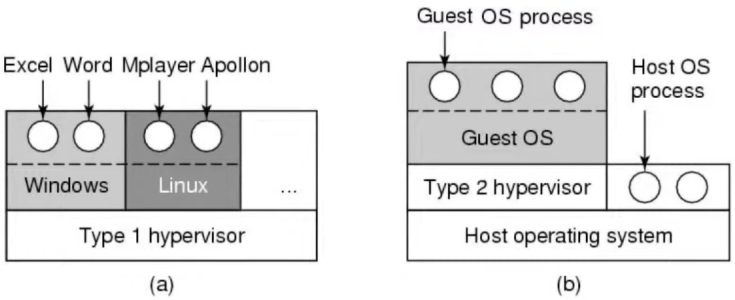
Application level virtualization

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.

8.6 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: type 1 and type 2



8.6.1 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.

8.6.2 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.

Question: In an emulator, if a VM has the same processor as host machine, will it be faster?
Answer: Yes, because the emulator will not need to emulate another processor and will default to full virtualization instead. Thus reducing the overhead and making it faster.

Question: Why would we want to start a VM with the same OS as the underlying host OS?
Answer: The VM can be used as a second machine for multiple reasons. For eg.: for testing purposes. However, the guest OS need not necessarily be the same.

Question: If the guest OS gets affected by malware, will it also affect host OS?
Answer: The host and guest OSs are separate OS environments on their own. Thus, if guest OS is bridged, it will not have an impact on the host OS.

Question: Why would we want to use Type 1 hypervisor and not install an OS?
Answer: If the requirement is to run multiple VMs on a single machine, a hypervisor should be installed. It will allow a large server to run multiple smaller servers each capable of running different applications. If a single OS is to be used on the machine, then virtualization is not needed and hypervisor should not be installed.

Question: Do Data centers use Type 1 hypervisors?
Answer: Yes, Type 1 hypervisors are very commonly used in data centers and cloud platforms.

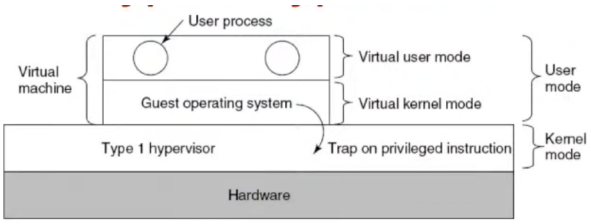
Question: Can one VM span multiple physical machines?
Answer: No, because an OS cannot run on different machines. However, multiple VMs running on different servers can communicate with each other to act like a distributed system.

Question: Do guest OSs communicate with each other via hypervisor?
Answer: Guest OSs have their own network cards with individual IP addresses and can thus directly communicate with each other over network. Since each VM is a software emulated version of a full PC, mechanisms such as shared memory cannot be used for communication between them.

8.7 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.

8.7.1 Type 1 Hypervisor

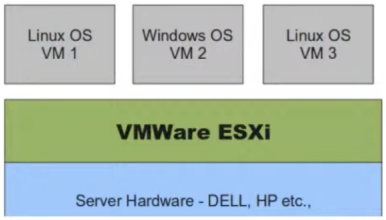


Theory: Type 1 virtualization is feasible if sensitive instruction is subset of privileged instructions or all sensitive instructions always cause a trap.
Reasoning: On booting a Type 1 hypervisor, it runs in kernel mode. A Windows VM run on the hypervisor should not be trusted as much as the hypervisor 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.

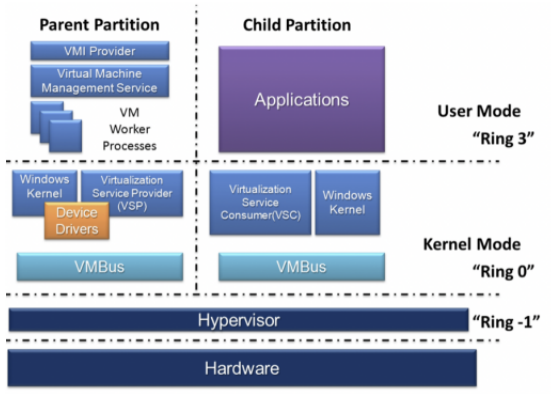
Early Intel processors did not have Type 1 support as they simply ignored any sensitive instructions executed in user mode. Recent Intel/AMD CPUs have hardware support, named Intel VT and AMD SVM. The idea is to create containers where a VM and guest can run and that hypervisor uses hardware bitmap to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

Question: Are there instructions that are privileged but not sensitive?
Answer: Yes. Eg: Divide by zero or an illegal pointer access. All these will generate an exception.

Examples:



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



https://en.wikipedia.org/wiki/Hyper-V

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. Linux OS has KVM loaded as a module. Thus, Linux OS is a hybrid OS which can be booted normally on a machine and can serve as a standard OS and also as a hypervisor simultaneously when KVM is enabled.

Question: In the above Hyper-V diagram, does the "Ring -1" indicate that the hypervisor runs with negative privileges?
Answer: No, there is no concept of negative privileges. It just indicates relative privileges.

Question: Can a physical machine run more than one ESXi hosts?
Answer: No, a physical machine can boot only one OS at a time, either a standard OS or the hypervisor unless there is hardware partitioning configured. But normal machines don't have that and can only run a single OS.

Question: Can Linux KVM be classified as Type 2 Hypervisor?
Answer: Since the KVM hypervisor is a part of the OS and not run as a separate application on top of the OS, it is classified as Type 1.

Question: How is a VM started in KVM? Do we boot host OS first or the VM can be started directly?
Answer: To start a VM, we always boot the hypervisor first. In Linux KVM, since the hypervisor is a part of the OS, we boot Linux OS first and enable KVM. VMs can be started on it now.

8.7.2 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 invoke the Type 2 hypervisor which in turn makes 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.

Question: How do applications that do not support VMs identify that they are running on a VM?
Answer: A VM simply looks like any other physical machine to an application. Thus, there is no technical reason as to why an application cannot run on a VM. However, some applications are not licensed to run inside VMs. Eg: MacOS. Applications can parse some markers to identify parameters such as drivers used by Type 2 hypervisors, but this is not a robust way.

Question: What happens if guest OS has hardware requirements that the physical machine does not satisfy?
Answer: This impact will be similar to when such a situation occurs in a non-virtualized environment. Eg: An OS might not boot if it has less resources.

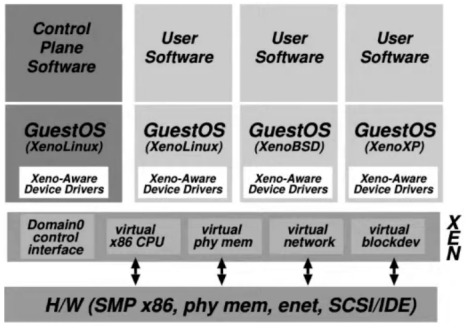
8.7.3 Para-virtualization

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. Since the modified OS eliminates all sensitive instructions with hypercalls, it can be run on any processor as hypervisor will handle invoking system calls. Paravirtualization is the only way to support Type 1 virtualization on older processors(which do not generate traps for sensitive instructions).
Note that such a modified OS will always need hypervisors and will no longer support standard non-virtualized environments.

Question: In a Type 2 hypervisor, after replacing sensitive instruction with a procedure, will it make a trap to the host OS?
Answer: It will not generate a trap, because the hypervisor will invoke a system call to directly ask the host OS to execute an instruction.

Question: What is the difference between a system call and a kernel trap?
Answer: A system call is essentially a function call to the OS through which which you can ask the processor to perform an operation whereas a trap is like an interrupt to the processor which the processor will have to serve by identifying what caused it.



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 or controller VM) just like HyperV. All OS drivers are placed in this VM(also called driver domain). So any other VM wanting to make a driver call will direct it to the driver domain via the hypervisor.

Question: Does this need communication between VMs?
Answer: Yes, but this communication is not network-based and can happen via hypervisor through IPC(Inter-Process Communication).

Question: Will para-virtualization have faster execution at the cost of having a modified OS?
Answer: It will definitely be faster than Type 2 virtualization.

8.8 Virtualizing Other Resources

8.8.1 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.

8.8.2 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.

Question: Does a virtual disk have 1:1 mapping or 1:N mapping to VMs?
Answer: Each VM will have its own virtual disk. Multiple VMs cannot be sharing the same virtual disk as theu might overwrite contents causing discrepancies.

Question: In a shadow page table, do we map guest physical address to host physical address?
Answer: Each VM has its own address space. Page tables inside the guest OS will keep mappings of this address space. However, this address space will translate to some memory addresses in the host space as well. The shadow page table holds these mappings.

Question: Does each VM have one shadow table?
Answer: Page tables are per-process. So each VM will have multiple shadow tables based on the processes running in them.

Question: Can Type 2 hypervisors leverage VT technology?
Answer: There can be a hybrid implementation of Type 2 hypervisor to leverage VT technology if the

underlying processor supports it. This reduces the translation overhead for sensitive instructions and thus make it faster.

8.8.3 Network virtualization

Similar to I/O virtualization, a network card is also virtualized. Here, VMs get a software emulation of the physical ethernet card. Whenever a read/write happens on this card, it creates a trap and the operation is redirected to the physcial ethernet card instead. The logical(emulated) network card cannot communicate on its own. It can only receive requests and route them to the physical card to process them.

8.9 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.

8.10 Use of Virtualization

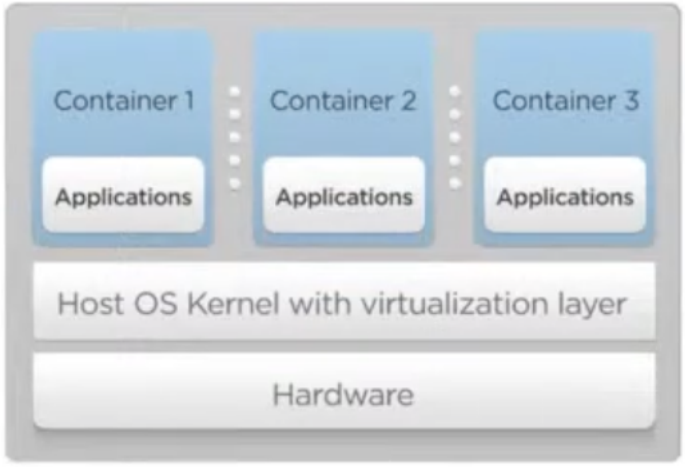
a) Cloud Computing b) Data Centres c) Software Testing and Development

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

Lecture 9:Virtualization

9.1 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.



Questions: Is Windows Compatibility mode also an example of OS virtualization?
Answer: No, it does not use virtualization. It involves more of driver versions handling to pretend like an older version.

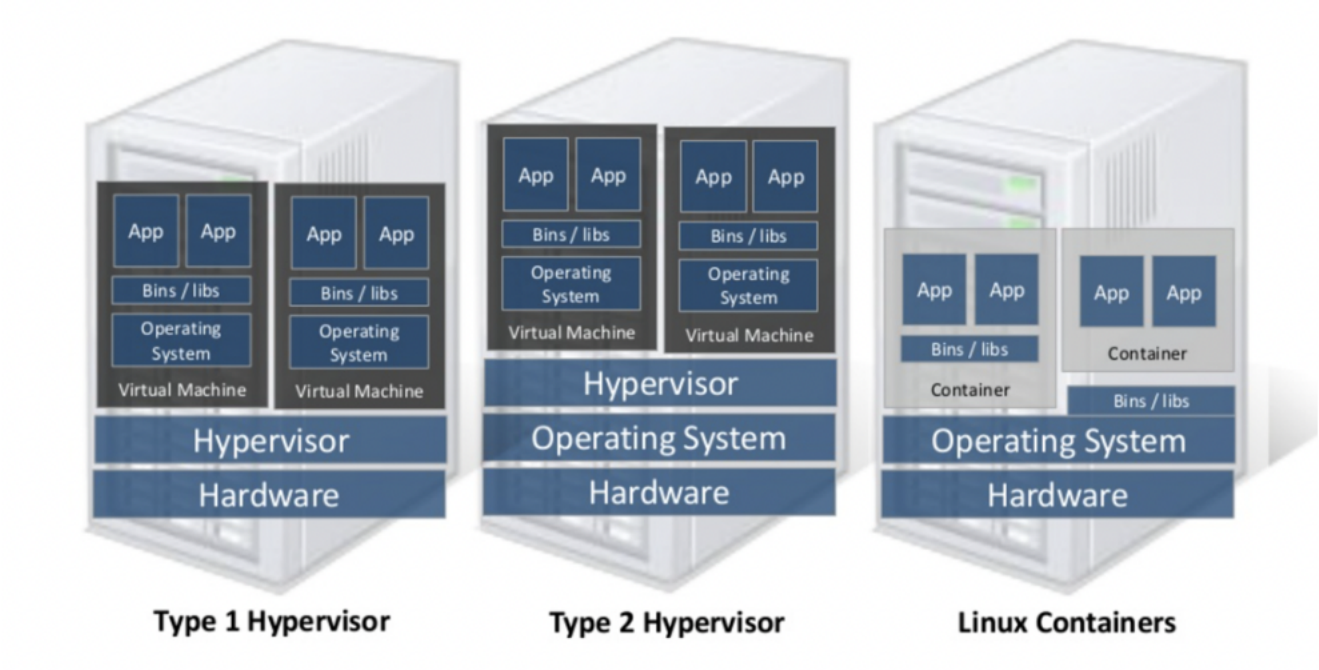
In OS virtualization, we create 'light-weight' virtual machines called containers. These containers are light-weight because they are built on top of the same underlying host OS. We cannot run an additional separate OS inside a container. Applications run directly inside containers and it uses the underlying host OS.

Question: Do we need to modify the host OS for OS virtualization?
Answer: The host OS needs to have support for OS virtualization, but it does not need any modification apart from that.

Question: What usecases does this have?
Answer: A base usecase for containerization is to islaote applications from one another. Docker is completely based on containerization.

9.1.1 Linux Containers(LXC)

Operating systems (OS) offer sandboxing and resource isolation capabilities. However, containers offer a more lightweight solution compared to virtual machines because they do not require an entire operating system to boot up. A container doesnot actually run an OS.



Question: In Linux containers do we have the ability to run multiple OS?
Answer: No we don't have the ability to run multiple OS on a linux container. In Linux containers we do not really run full pledged OS. We can emulate an OS interface inside the container.

9.1.2 OS mechanisms for Linux Container

Namespaces

Namespaces can allow control over what resources are visible to a process. Resources such as processes, mount points, network interfaces, users etc. Eg: If you restrict the processes that a container can see, if a user runs ps or top command on the container, it will only return a subset of the processes that are actually running on the physical machine.

Question: How is a sandbox different from a container?
Answer: The terms are being used interchangeably. Container is a technical term.

Cgroups

Cgroups or 'Container Groups' can allow control over how much resources a container can consume. We can control/limit the amount of CPU allocated, disk space allocated to a container. Eg: A container be allocated 1 GB of memory. Thus, all processes running on the container can collectively consume only 1GB of memory and anything above that will start failing. Resources such as memory, CPU cores, I/O cycles, network bandwidth etc.

Question: Can resources allocated to a container be changed on the fly?
Answer: Yes, utilities available for that

9.2 Proportional Share Scheduling

This is a policy used to allocate resources across processes where-in each process is assigned a weight and it is allocated resources accordingly.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. However, if the container is idle and is not using the resource, its share is redistributed across other containers/processes that have something to run in proportion to their weights (fair share scheduling).

Lottery-based scheduling: Fair share scheduling in a randomized way. We assign each container some number of lottery tickets. The CPU scheduler decides which process/ container gets the next time slot via a lottery – it's going to take all the tickets that have been assigned to all the containers and pick one winning ticket randomly. The container that holds the winning ticket gets to run next next time slice. The number of lottery tickets a container has determines the chance of winning the lottery. So, by controlling the number of lottery tickets, we can decide how much share of a resource a container will get.

Hard limits: Hard limit means there is a hard allocation. Even if the container is not using the resources, it is not redistributed across other containers. Those cycles are simply wasted.

9.2.1 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 $s_i = \max(s_{min}, s_i + \frac{q}{w_i})$.

Question: How does the scheduler know whether the container has used CPU cycles?

Answer: Let's only consider processes first. Each process gets a weight, and the share-based scheduler schedules these processes. These processes get CPU share in proportion to their weights. The CPU scheduler knows which processes are blocked on I/O and which processes are ready to run. If a process is blocked on I/O then it is not running and basically giving up its share. If some process is runnable, that means it is active. It will get to run and its counter will be incremented. It's the same concept for the container. There are processes in the container, so if any process from that container runs that container's counter is incremented based on the cycles used by that container. If there's nothing running, then the counter doesn't increment and something else gets to run.

Question: Is the number of processes static or dynamic?

Answer: The number of processes/containers are assumed to be dynamic. It will change over time. It means that the relative share of the resource you get will change over time. Say a container is given a weight of one. If there's nothing else in the system, it means that the container gets a hundred percent of the resource. If another container gets created and it also gets a weight of one, now the resource is shared 1:1. If a third container comes now again the allocated fractions will change.

Question: In practice who is managing the weight?

Answer: The user manages the weight that is given. If nothing is given it just assumes a weight of 1.

Question: Why is $s_i = \max(s_{min}, s_i + \frac{q}{w_i})$?

Answer: This says that if a process P becomes inactive and then comes back and becomes active, its counter has to be at least the min counter of the system. This is because its counter was the same for a while and so it's likely to be the least, as everyone else's counter has advanced. Now, if I simply pick the min=1, P will start hogging the resources. So, whenever a process becomes active its counter has to be at least the minimum counter in the system, and then from that point on, its counter is incremented. If a new process arrives and there are existing processes running its counter can't be initialized to 0. Its counter has to be initialized to the min counter of any active task in the system. That way it is at least starting as the minimum task in the system and not 0.

Question: Is the value of s_{min} going to always increase?

Answer: Yes. It's basically tracking the process/container that has received the least service (least amount of cycles in the system). So, it will increase monotonically.

9.3 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 the 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 containers on other platforms, as it would have to translate Linux calls to other platform calls (Windows/Mac Calls).

Docker is basically a layer around the Linux container (refer Fig. 9.1) that provides tools to build these containers very easily and distribute them. For example, we can distribute them through git repositories. We can just download a container from a git repository and run it on any machine. Docker images will run anywhere Docker runtime is present. All we need to do is download a Docker image and just say run. We don't have to create a container or configure it (e.g., allocate resources). Docker has done all of it.

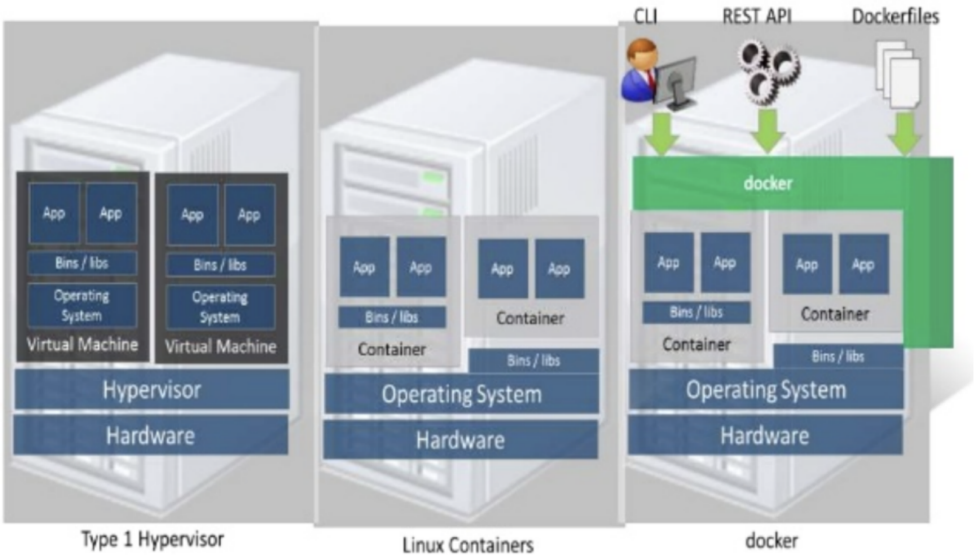


Figure 9.1: Visualization of type 1 hypervisors vs Linux containers vs docker.

Docker is a Linux-based OS virtualization technology. However, it can run on any platform (e.g., Mac OS). This is because there is a hidden version of Linux that Docker is starting which is not visible to us.

Question: Is Docker a type 2 hypervisor?

Answer: Docker is a container management runtime. So, it is not technically a type 2 hypervisor. It is using a type 2 hypervisor to run Linux in the background, and then it's running on that Linux.

Question: Is Docker performance the same in all kinds of systems?

Answer: If we are running Docker in any virtualized environment, there will be some overhead which is not going to be present if we are running it in a native environment. For example, if Docker is running on native Linux, the overhead will be less than if it is running on a Linux VM that's running on a Mac.

Question: If you have multiple containers running, do each of them run on a separate VM?

Answer: All of them run on a single Linux instance. if the machine does not have native Linux it will create a virtual machine with Native Linux and all the containers will run on that machine.

Question: If you create a virtual Linux can you go into the shell of the Linux?

Answer: The docker prompt that was visible during the demo was a shell on that Linux machine. It's just that it's inside a container.

Question: To create a python image, you cannot enter the shell; but if you do Ubuntu, you can enter.

Answer: That's not the case. All containers, regardless of what is packaged in them (say, python package), run on a Linux instance. So, we will get a shell, which is a shell for the container. The application is sandboxed in the container.

Question: In the language image, we cannot enter the shell.

Answer: That's not the case. Whatever the language runtime is, it is still running on that virtualized Linux

instance. Even though we showed a web server package in the demo, we still haa a shell that is on the native instance. If we do "top", we will see our process(es) running. The Linux instance may have other running processes, but we don't see everything as we did a namespace and hence we can see only the processes that are running inside our container.

Question: Are namespace and containers the same?

Answer: No. A namespace is a way by which we can limit what a process can see. A container is a namespace + C-group (which allocates some resources to those processes) + some other things. A container has a namespace associated with it, but a namespace itself doe not make a container.

9.3.1 Docker Images and use

Docker uses a special type of file system called as a union file system (AuFS). Docker assumes that there are a base set of files that are already present on the Linux OS and if the container needs the same files then the existing set of files on the OS are used. If they are not already present the files are bundled with the docker image. The container images are made compact due to this process.

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

Lecture 10:Proportional Share Scheduling

10.1 Migration Introduction

The motivation behind developing techniques for code, process and VM migration is that migrating these components of a system helps improve performance and flexibility. For example, in distributed scheduling, we submit a job on one machine. However, if that machine is overloaded, we would want to migrate either the code (of the submitted job) or the process itself to some other machine and improve performance. From a flexibility standpoint, migration helps in the sense that we can configure distributed systems dynamically. For example, clients can download software on demand from some driver repository and don't need the software to be preinstalled (refer Fig. 10.1).

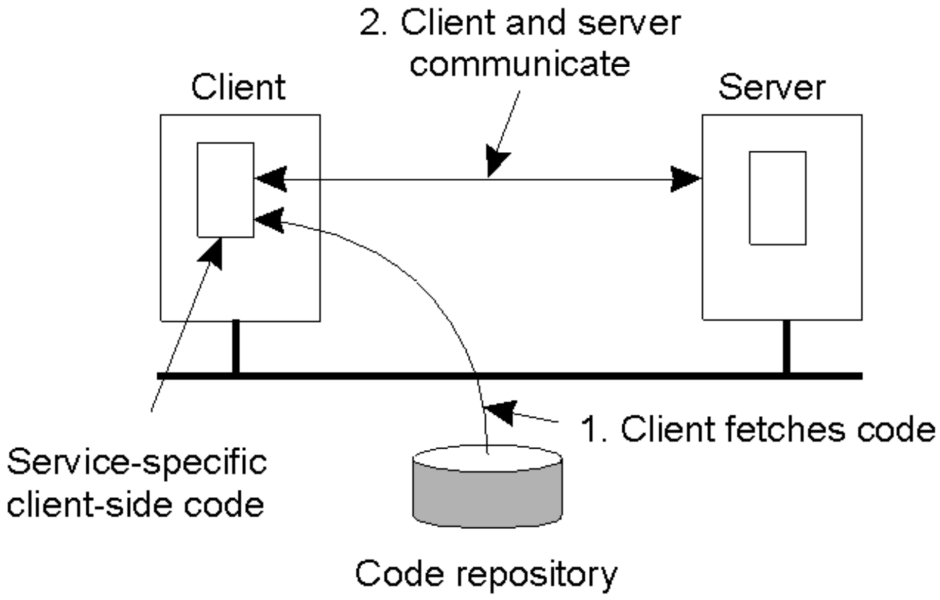


Figure 10.1: Motivation for migration – adds flexibility.

There are two types of migration models:

- **Process Migration (aka 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 (aka 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. Many scenarios map to code migration. For example, filling out a web form and hitting submit is a form of code migration because the form becomes a small piece of code that goes to the server, which then processes it and executes. Another example is doing a web search. Web search can be thought of as a query for some words, and we want all the pages for that query. So, that's a "program" we made. The query actually moves to some other system (in this case, a search

engine) and is executed there. Simple examples of code migration are taking a real Java program or a binary, moving it and executing it in another machine. Docker is also considered to be an example of code migration. Additionally, anything that is "download and execute" is also an example of code migration — for example, downloading device drivers on demand for new hardware.

We can use code migration to get better parallelism (replicating same code across machines).

Question: When you do code migration, are you doing process migration?

Answer: Code migration simply means moving files, not processes. For example, Python code is a file. If we want to execute it remotely, we will copy that program somewhere else and run it. The process is created when we start executing that code on a remote machine.

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.

10.1.1 Migration models:

Migration can be sender-initiated or receiver-initiated. An example of receiver-initiated migration is a browser downloading a Java applet or Flash application from the server. In sender-initiated, the sender process has the code and sends it somewhere else. An example of sender-initiated migration is a web search and database query. Fig. 10.2 shows a flowchart of the migration models.

A process can be migrated or cloned. In the 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 we kill the process in the sending machine and start executing the process in the receiving machine. We can also allow both copies to execute. Cloning is a convenient way of replicating the process. An example of cloning is forking a process.

Question: What does it mean to "execute at target process" or "execute in separate process"?

Answer: Let's take an example to understand this. Say, we have a browser that contacts a server. The server sends the browser a small piece of code (say, JavaScript). After migration, if the Javascript executes in the same process of the browser, then it is "execute at target process". In contrast, if the browser process created a second process and Javascript was being executed in that second process, it is "execute in separate process".

Question: Whose decision is it to decide whether to do it in the target process or a separate process? Is it the receiver or the sender?

Answer: Typically it's neither. It is the developer's decision (application-level decision).

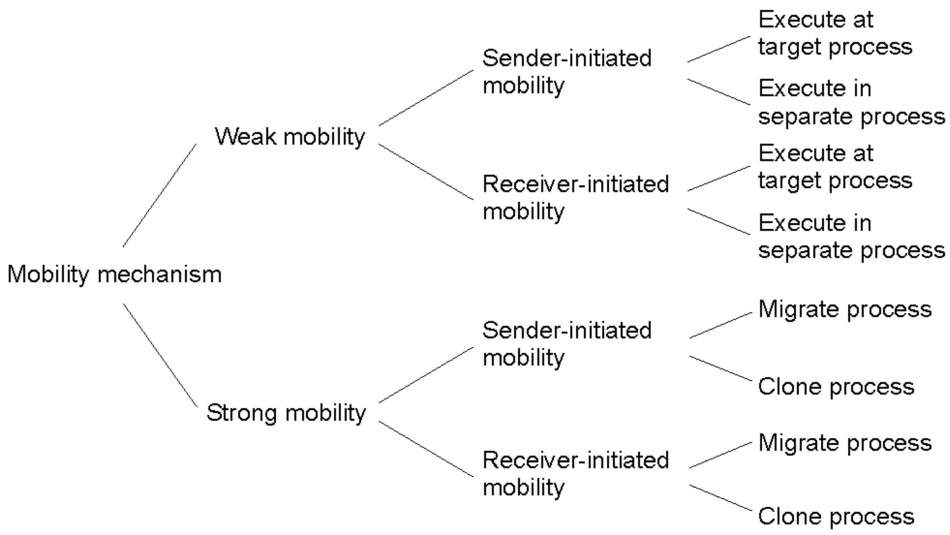


Figure 10.2: Migration models.

10.1.2 What happens to the resources that the process was accessing?

Let's say a process migrated to a remote machine was accessing a file on the local machine. The process needs to read/write to the file, but it is still in the previous machine. If the file is not present on the new machine, the process will throw an error. So, we need to deal with all kinds of resources that the process was accessing when we deal with code or process migration. How we are going to handle resources depends on what type of resource it is. The process must continue to have access to the resources even after migration to avoid any errors.

We classify the resources along two dimensions. To decide whether to migrate a resource attached to a process or not, first, we look at the nature of binding of resource to a process. There are three types of resources to process bindings:

- **Identifier:** Hard binding, one that you cannot substitute. Least flexibility. An example is URL for a website.
- **Value:** Slightly weaker binding. We can substitute it with another similar resource. Libraries used in Java are a good example. If the Java process was using JVM in the previous machine, as long as the new machine has a JVM of the same (or compatible) version, the Java process will work fine.
- **Type:** Weakest binding. We can substitute it with another resource which need not be exactly similar. Maximum flexibility. An example is a local device like a printer — different printer models will work just fine as long as there is a printer ready to accept print jobs.

Second, it is also necessary to look at the cost of moving resources. This can also be classified into three categories:

- **Unattached:** Very low cost of moving. E.g., files.
- **Fastened:** High cost of moving. E.g., databases. Databases can be moved, but if their size is large, moving them is expensive (in terms of time, or, network bandwidth cost, etc.)
- **Fixed:** Can't be moved. E.g., an ethernet card or an IP address.

10.1.3 Resource Migration Actions:

Different combinations of resource-to-machine binding (columns) and process-to-resource binding (rows) 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 (resource can't be moved, so giving it remote access), MV means moving the resources, CP means copying the resource and RB means rebinding process to locally available resource.

10.1.4 Migration in heterogeneous systems:

Here, the assumption is that the sender and receiver machines are heterogeneous (different hardware/OS etc.). If the OSes are different (say, Windows and Linux), we cannot just copy the memory of a Windows process to the Linux machine and expect it to run, due to different binary formats in the different OSes. If we have different hardware then the instruction sets will be different (say, Intel vs ARM). Thus, process migration is much harder due to these constraints. Code migration is easier. For example, as long as Python is installed on both machines, a Python program written on a Linux machine will run fine on a Mac.

Only if we can abstract out the differences in OS/hardware, we can migrate processes. For example, Java. Java runs on JVM which is platform independent and abstracts out the OS/hardware differences. So, a Java process running on Windows can still run on some other platform.

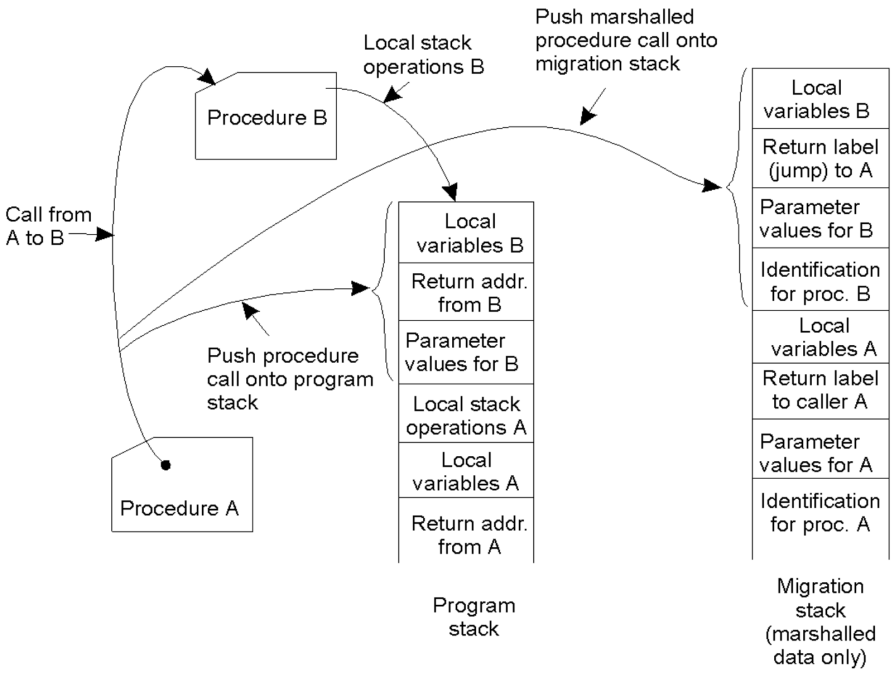
Question: When you get a Docker image is that process migration or is that code migration?

Answer: If we take a Docker image and start it, that's code migration because we got the code for the program. It's just packaged in a container. On the other hand, if we have a running container and we move that running container to another machine, that's process migration because the container actually has active processes running.

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.



10.2 Virtual Machine Migration

If a process is communicating with some other process, then process migration is much harder as the IP addresses of the sender and receiver machines are different and we cannot move IP addresses across physical machines. Virtual machine migration (VMM) gives us a way we can actually deal with even network connections of a process. Even the network connections will move when we migrate a VM. The IP address will also move and it will not change.

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 downtime. 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 which are Pre-copy migration and Post-copy migration.

Pre-copy Migration: Fig. 10.3 shows pre-copy VMM (A and B are two physical machines). 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

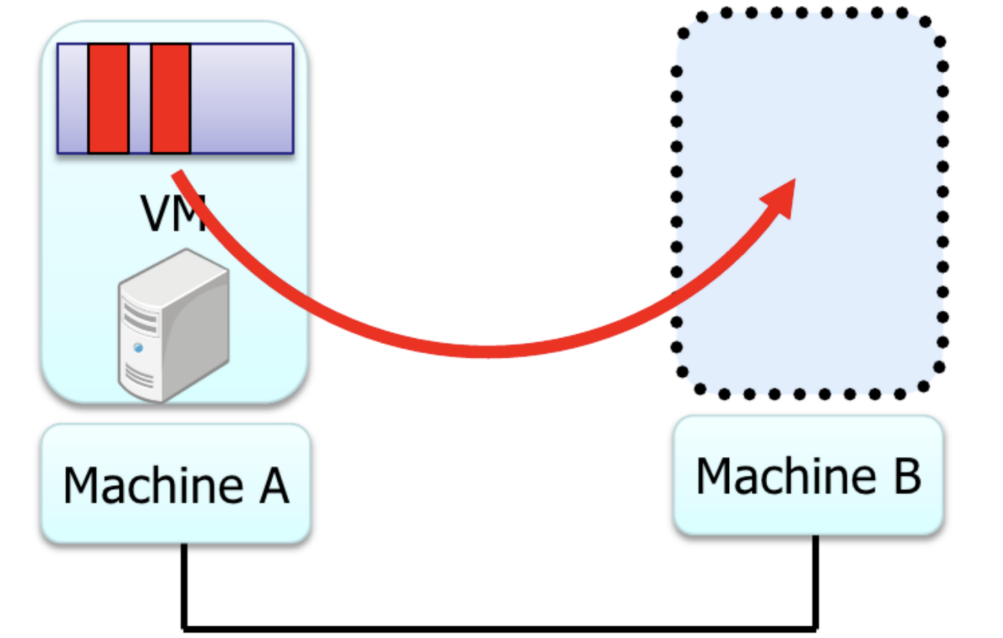


Figure 10.3: Pre-copy VMM.

Question: Is it a fair operation to assume that copy operations are faster than memory updates?

Answer: No, memory updates are much faster than copy operations.

Question: When copying in an operating system, is the entire thing is updated again?

Answer: In operating systems, there's a concept known as the working set of processes, which refers to a set of memory pages actively used by a process. Despite a process being allocated a significant amount of memory, it typically only accesses or modifies a small portion of it at any given time. Therefore, when copying data, it's not necessary to update the entire memory space; only the portions that have been modified need to be copied. In essence, only the relevant memory pages are changed during the copying process, rather than the entire memory space.

Question: What happens in 2nd round if a page which was not changed in round 1 got changed?

Answer: In the 2nd round, that page will also be considered. Our main goal here is to select all the pages which have a dirty mark on them and send them to another machine. However, it's worth noting that our assumption remains valid as the number of modified pages typically decreases from round to round.

Question: How do we qualify the downtime?

Answer: From the application perspective there is a downtime in this process but this is very small may be in milli-seconds which is not perceivable from the user's perspective.

Question: What happens if the page writes are many?

Answer: Despite the potential for many page writes, ultimately only a single page is changed. Therefore,

the number of pages changed is not affected by the frequency of page writes. Consequently, there should be no issue with this pre-copy method. However, if there are random writes to different pages, the assumption that our converging changed pages process holds true is invalidated, rendering the process ineffective.

Question: Is there any synchronization here?

Answer: No, synchronization is not involved in this process. We are simply moving the memory content while keeping track of changed pages and moving those pages in the second round, continuing the process further.

Question: When a page changes are you going to send the entire page or are you only going to send what part of the page that changed?

Answer: In this case we'll only track dirty pages and send the entire page. If we want to send only the diff, then we have to keep the old copy and the new copy and do a difference, and that means even more memory allocation and added complexities.

Question: Till when do we repeat Step 4 above?

Answer: We assume that we send all the pages the first time. Before sending again, only a fraction of the pages change. Since it is a subset, it will take less time than the previous round, as the network bandwidth is fixed. This is true for all rounds; that is, each round takes less time than its previous round (assuming no adversarial application). Eventually, after n iterations, we will have a small amount of memory that we need to move. At that point, we go to step 5, that is, stop the VM and copy the remaining pages to the destination physical machine. Since we stopped the VM, no new pages were dirtied during this round.

Note that although it says "live" migration, there is a small time during when the VM is stopped.

Question: What is the threshold where the stop time is small?

Answer: That's configurable. We want the pause time to be as small as possible so that it's almost not noticeable. So, it will depend on factors like the network bandwidth between the two machines.

Question: What happens if there's a network failure?

Answer: If anything fails, VM migration will stop, and it will not continue.

Handling "IP migration": The host machines have different IPs. This is why process migration doesn't work. However, VM migration works because now there are two different IPs — VM IP and host IP — and we can actually keep the VM IP the same while migrating the VM. The applications running in the VM are coupled with the VM IP, and hence these connections do not break even after VM migration. For everything to work after migration, there is one last step — an ARP packet is sent to the ethernet switch to update the ARP-MAC mapping (although IP remains the same, MAC address has changed since physical hosts are different).

Question: Is the IP address is the VM public or private?

Answer: It doesn't matter. The applications are actually tied to the VM IP address and not the host IP address. The only thing is VM IP should be different from the host IP (source/destination host). The application connections will not break as VM IP remains the same.

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. Here, we have a trade-off between memory overhead and performance overhead. 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.

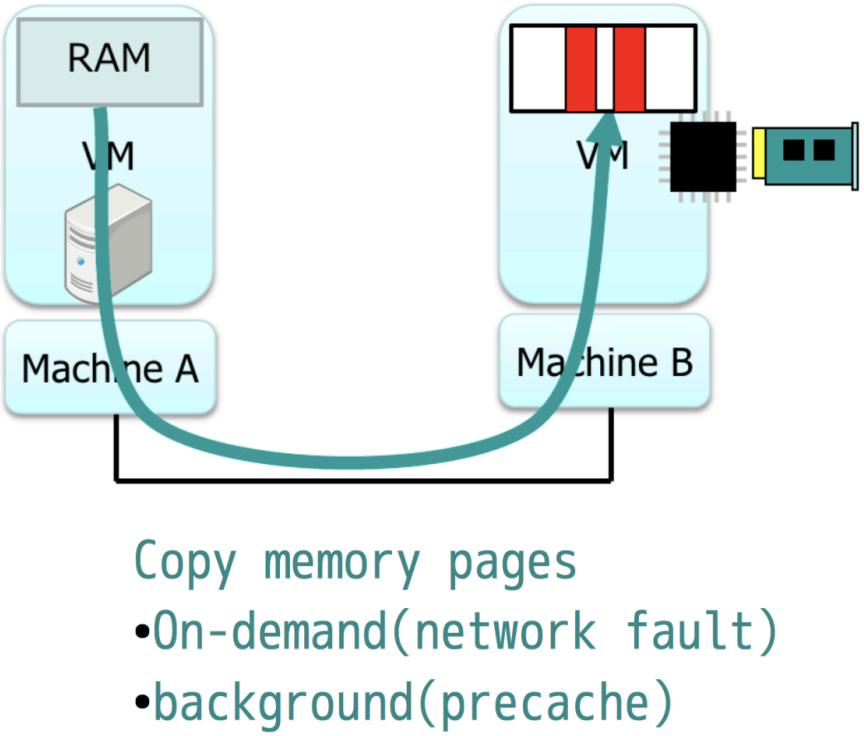


Figure 10.4: Post-copy VMM.

Question: Can a Machine B be part of different network or do they have to share a switch ?

Answer: This process works on one cluster.You are not migrating In this process something on the internet to some completely different location.that is WAN. What here is LAN migration(Local Area Network).So here it assumes that Machine A and B are part of the same LAN.Presumably they are part of the same subnet. They don't have to be on the same switch. In WAN,In a complete different address space,it won't work since routers have to send packets to the right port.Only works for Local Area Migration.

Question: What about the files which are on the disk?
Answer: files are storing in virtual disk. 1)virtual disk image is stored on the file server. Can simply access from the file server from a different machine.Most common and fast approach. 2)virtual disk image is local to machine A. We need to move the virtual disk as well. Copy the file page by page by precopy mechanism.Slow process.

Question: Given that it is happening on cluster, Is there any restriction on hypervisor or OS?

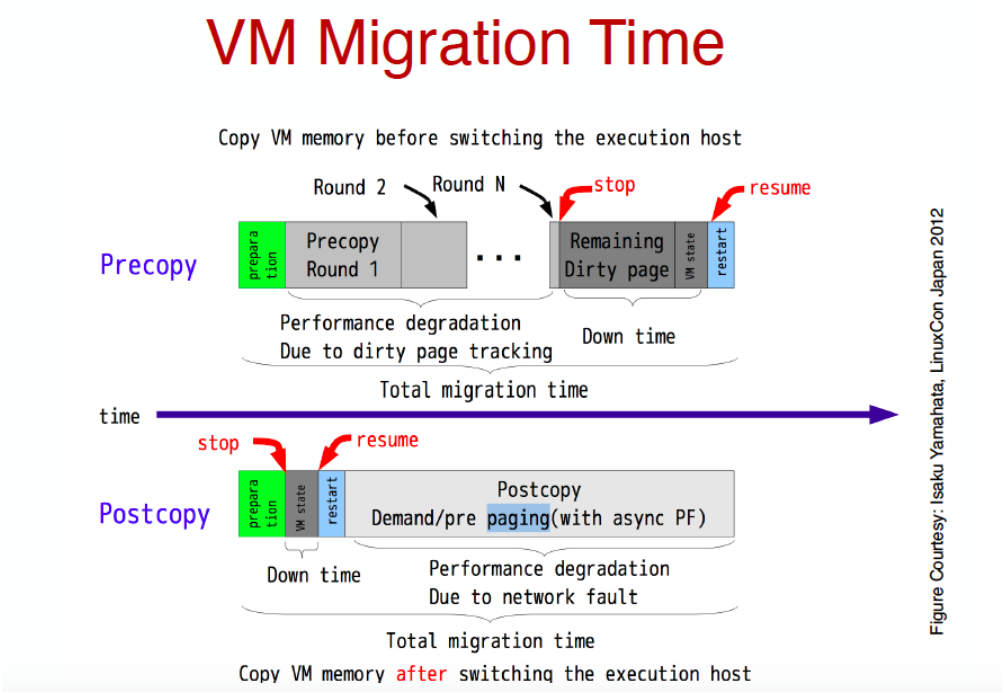


Figure 10.5: Visualization and comparison of pre-copy and post-copy VM Migration.

Answer: Only restriction is VM should be executable on the new hypervisor.We assumes that machines are running the same hypervisor.

Question: What are the pros and cons of precopy and postcopy migrations? **Answer**: In post copy,since you are fetching memory on demand,Its gonna take some time to fetch each page since the process which got faulted when it was trying to execute and access that page gonna pause untill that page is brought over.So it causes some performance penality.But each page is transferred exactly once.

In precopy, there is no performance penality.But the pages has to trasnfer multiple times if it has changed after being copied over.The total amount of data copying in precopy is going to be much higher.Also there is a small chance that precopy may not terminate,especially when the VM behave badly,it keeps churning memmory and after each round you have transfer a huge amount of memory and it never converge.

Question:Why do you have to tranasfer the VM?

Answer: Vm can allow you to increase the resources When your application get overloaded.So load balancing is a good reason.

Question: In post copy ,can there be a scenario where some pages are never accessed by processes and no page faults?

Answer:To prevent these scenarios, there are 2 ways we are bringing memory, 1)Fault on Demand 2)A background process which is fetching remaining pages even if they are not faulted on.

Question:In post copy ,are files migrated in the same manner?

Answer:you can assume the files are on the server and don't move it at all. or can assume that files are on the virtual disk local to the VM and either move it over through postcopy or precopy. easiest approach is keep things on the file server and don't move.

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.

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.

10.3.1 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.

10.3.2 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: Is the snapshots are periodic diff?
Answer:Periodic diffs are not necessarily copy and write. copy and write is diff on demand.You preserve the copy of the page and then you write.

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.

10.3.3 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.

10.3.4 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.

10.3.5 Case Study: Viruses and Malware

- Viruses and malware, classified as code migration, propagate between machines, often initiated either by the sender or the receiver.
- Sender-initiated threats, such as proactive viruses, actively seek out vulnerable machines to infect, deploying autonomous code to accomplish their objectives.
- Receiver-initiated threats occur when users unknowingly trigger malicious code by interacting with infected web URLs or opening email attachments containing malware.

Lecture 11:Virtual Machine Migrations

11.1 Datacenters

A datacenter is a facility where a large number of servers along with lots of storage run multiple applications(server farm).

A common way to image a Datacenter is a supermarket where shelves are composed of servers.

Datacenters generate a lot of heat and they need cooling infrastructure. They consume lots of power too(electricity).

11.1.1 Architectures

- Traditional:
  - Applications run on physical machines.
  - Manual management of server by System admins.
  - Uses SAN and NAS to hold data.
- Modern:
  - Applications run on VMs.
  - Uses Type1 hypervisor.
  - Pre and post copy migrations is done on VMs.
  - VMs can be resized and migrated.
  - Allows automation.

11.1.2 Virtualization in Data Center

- Virtual Servers Example are consolidated servers where you can replace N old servers with 1 new server .
- Virtual Desktop
  Example: PC in the Cloud - In an enterprise, rather than giving powerful hardware to the employees, employees get a thin client that connects to a powerful VM in the cloud.

Question: How migrate-able is GPU as a resource?
Answer: We can virtualize a GPU and slice it into virtual GPUs (for example, NIVIDIA MIG) but this is fairly recent and can be expensive. Old GPUs do not have this feature and have to be used fully.

PUE (Power Usage Effectiveness) = Total Power/IT Power

typically PUE is 1.7
Google's Datacenter has PUE about 1.1

11.2 Cloud Computing

Cloud Computing is where servers and storage are going to be leased by the customer. Cloud providers allow you to rent on demand, not only on an advance, pay-as-you-go model.

Benefits:

- Remote access from anywhere
- Pay as you go
- Highly scalable

Question: Do cloud providers have the ability to handle peak loads?

Answer: They have the instinct to develop the most efficient data centers.

11.2.1 Types

- Infrastructure as a Service (IaaS) : Machines/VMs with network access
- Platform as a Service (PaaS): Platform to run the applications by the provider
- Software as a Service (SaaS): Hosted application by the provider.

Spot Instances - Excess capacity is rented at a high discounted price. However, the access can be revoked by the provider with a very short warning. Part of IaaS.

11.2.2 Cloud Models

- Public - The cloud provider owns the resources. Enterprices suspicous of the security.
- Private - Org builds its own private cloud.
- Hybrid - Exceess private load is put into public clouds.

11.3 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)

11.4 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.

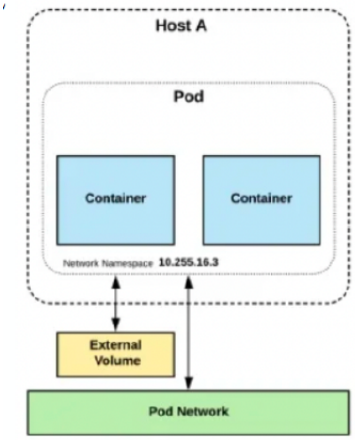


Figure 11.2: Visualization of the relationship between containers, pods, and hosts.

11.5 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.

Lecture 12:Kubernetes

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. Just like the drift developed over time in a mechanical clock due to the quartz crystal inside it, computer clocks also develop drift over time since they also use a chip which depends on a physical material to tell time. This drift needs to be calculated to determine how frequently synchronization is needed.

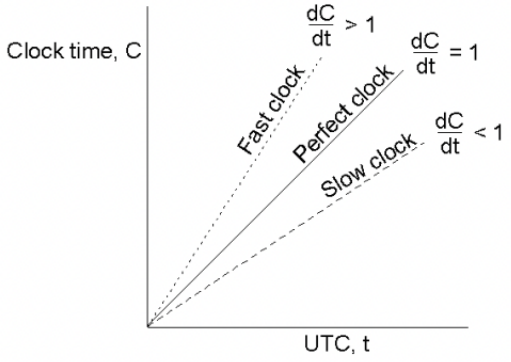


Figure 12.1: Clock drift relative to a perfect clock.

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 maximum drift rate 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. Two clocks may drift by $2\rho\Delta t$ in time Δt. 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.

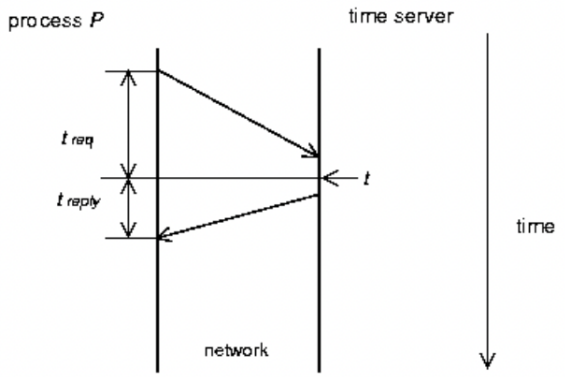


Figure 12.2: Cristian's Algorithm.

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.

Question: Does the drift rate change over time?
Answer: It may change over time, but the manufacturer of the clocks guarantees that the drift will be between 0 and ρ (both included).

Question: What happen to a processor when we change the value of a clock?
Answer: Processor does not care about the time, we are just changing the time, not what is happening at the processor level.

Question: What if t_{reply} is greater than the re-synchronization?
Answer: That should not happen. Clocks should not be that off. Re-synchronization should happen every seconds or minutes granularity, but propagation delay is in milliseconds.

Question: What happens if the clock is advanced? backwards?
Answer: When $t_{reply} - t_{send}$ is positive, we cannot tell that the clock is advanced. However, the negative value of $t_{reply} - t_{send}$ can cause the problem to the system.

When the clock is advanced, we cannot tell from the $t_{reply} - t_{send}$ that the cloc

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. The latency caused by the network while communicating is also adjusted during this time at the receiver end. 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. Note that the clocks can still drift from each other, but not more than 50ms. This also means that the estimate of the network roundtrip time cannot be off by 50ms.

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}), using eigth pairs of delays from A to B and B to A. 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. Note that different hierarchies can synchronize with each other with 50ms maximum error. This means that in a larger organization, only one time server is needed to synchronize with other computers.

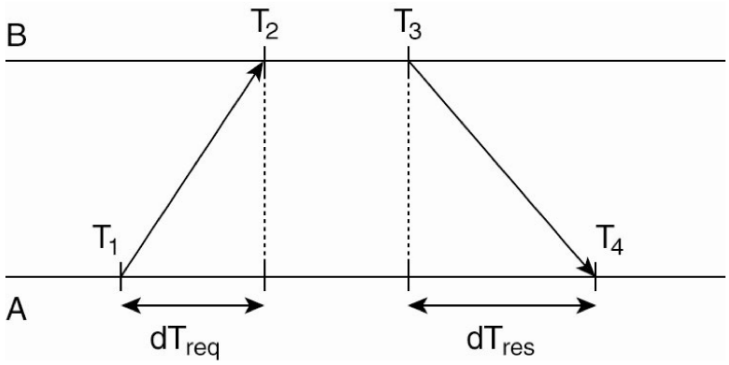


Figure 12.3: Network Time Protocol (NTP).

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. It is assumed that the satellites are in perfect synchronization with an atomic clock.

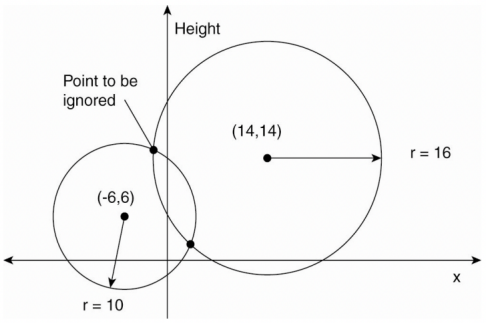


Figure 12.4: Global Positioning System (GPS)

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, the satellites just keep broadcasting the time. Note that you have to have the geo-stationary satellites for this to work (move at the same rate as the earth).

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. In other words, the logical clock allows us to reason the order of the event, but not the actual time difference between the events.

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.

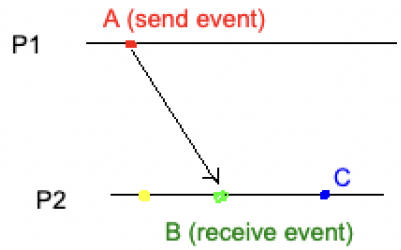


Figure 12.5: The "happened-before" relation. Note that we cannot say anything about the relation with the yellow event.

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.

Lecture 13:Clock Synchronization

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.

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.

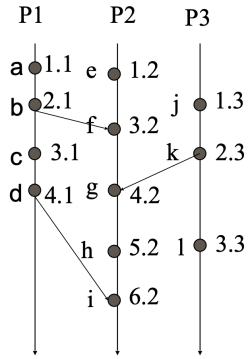


Figure 13.1: Creating total order by appending process id

13.1.3 Example: Totally-Ordered Multicasting

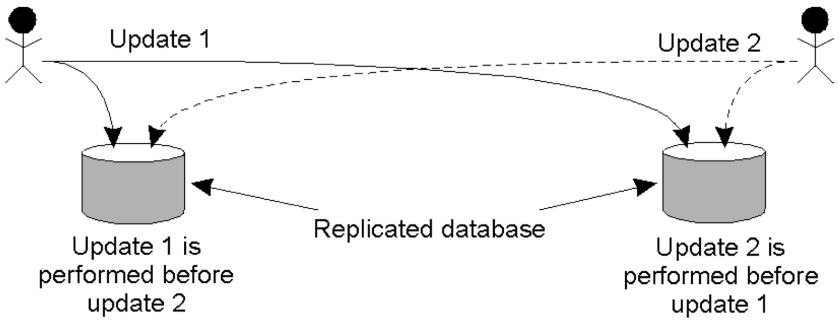


Figure 13.2: Example: Totally-Ordered Multicasting

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.

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.

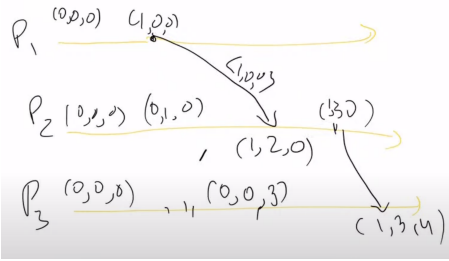


Figure 13.3: Vector Clocks

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.

Q: Are the clock distributed among each process?
A: The clock is not distributed, it is a vector that each process maintains (i.e., each process maintains its local array)

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.

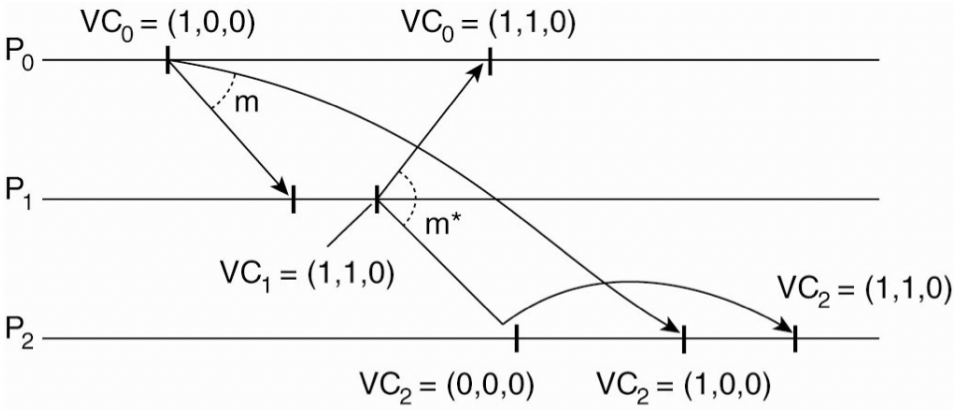


Figure 13.4: Enforcing causal communication

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.

Q: Can we instead send messages with unique IDs, so that we can track which message is sent/received?
A: You could, but the consistent cut eliminates the need for the unique IDs.

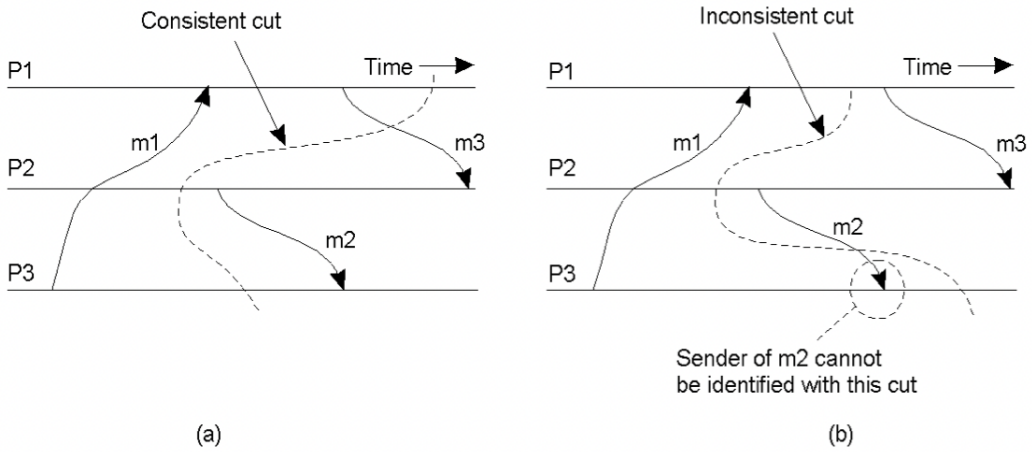


Figure 13.5: Consistent and Inconsistent cuts

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.

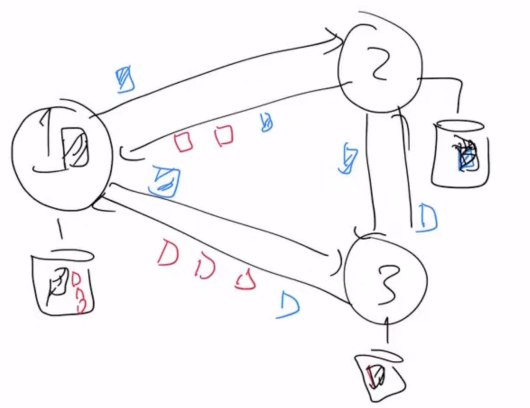


Figure 13.6: Distributed Snapshot Example

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.

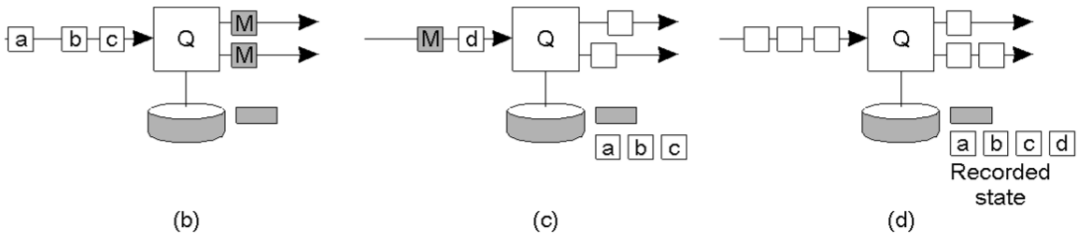


Figure 13.7: Snapshot algorithm example.

Q: If you have N processes in the system, does that mean each process has to wait for $N-1$ markers before it can stop recording?
A: The number of the expected markers is not dependent on the number of processes in the system. Instead, the number of expected markers of each process depends on the number of socket connections of that process.

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.

13.4 Election Algorithms

13.4.1 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.

Lecture 14:Total Order

14.1 Overview

This section covers the following topics:

Distributed Snapshots: Distributed Snapshot Algorithm

Leader Election: Bully Algorithm, Ring Algorithm

Distributed Locks: Centralized, Decentralized, Distributed algorithms

14.1.1 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. This is done so that in case one or more components of the distributed system fail, the system as a whole can restart from a snapshot of all the components rather than starting from scratch as that would waste all the CPU cycles that have already been consumed in the processing before the system failed.

14.1.2 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 its other channels. If the process sees a marker for the snapshot at the second time, it will stop saving messages for the corresponding 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.

Question: Can the leader be a single machine?
Answer: We don't actually care about the number of machines in the system. Here, it is a process that is chosen as the leader, regardless of the number of machines and the number of processes running on each machine.

Question: Are we assuming here that every node knows about every other node in the system and that they can communicate with each other without a middle-man?
Answer: Yes, here we assume that all processes are aware of the presence of all other processes and can communicate with each other.

Question: Does the chosen leader also need to know about the presence of all other processes in the system?
Answer: That depends on the type of work that is expected out of the leader. For instance, if it is a time-server, then it only needs to answer time queries and not anything about other processes but if the leader's job is distributed lock synchronization, then it needs to know the state and other information of all the other processes. Responsibilities of the leader are not a part of the leader election mechanism.

Question: How do you tell whether a process (leader in this case) has really failed/stopped or is just slow? *Answer*: There is no way to differentiate between a slow and a stopped process as such. So all these algorithms work based on time-outs. The process is declared to be dead if it is slow enough to respond after the time-out. Essentially, a failed/stopped process is equivalent to an infinitely slow process. However, when the old leader (slow process) discovers that another leader has been chosen and their process ID is smaller, the old leader can reinitiate the leader election process.

14.1.3 Ring-based Election 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 know about the new coordinator.

Note that a process might have to jump over its neighbor and contact the neighbor's neighbor in case the neighbor has failed, otherwise the circulation of the election message across the ring will never finish.

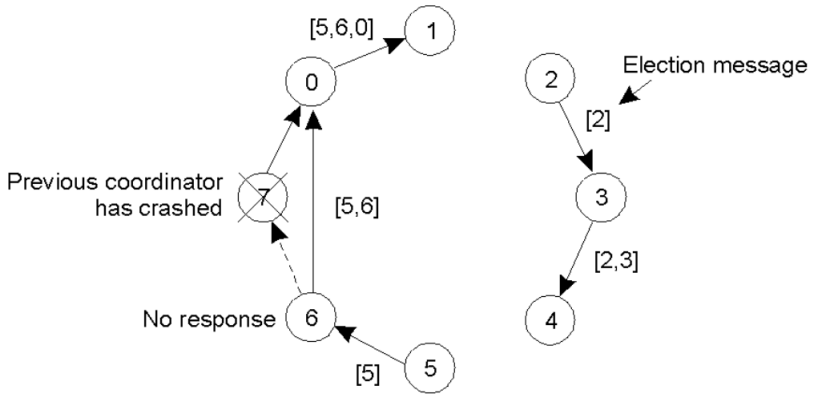


Figure 14.1: Depiction of Ring Algorithm

An example of Ring algorithm is given in Figure 14.1. 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: What initiates the election?
Answer: Any process can initiate the election. Say the leader L was a time-server and a process X discovers that X is not responding anymore, then X can initiate the election process.

Question: Is the ring topology applied only for elections?
Answer: Yes, as this is a distributed system, communication between any pair of processes can take place. But for the purpose of leader election, a process of ID i will only communicate with the processes having id $i-1$ and $i+1$, i.e. as a ring topology because this brings the complexity of the leader election down to $O(n)$.

Question: How many members of the ring does a process need to know?
Answer: In the worst case, a process will have to communicate with all members of the system during the election process, i.e. in a scenario when most of the processes have failed and stopped responding, the remaining processes will have to jump over multiple processes in between in order to elect the next leader.

Question: If every node knows the process ID of all other nodes in the system, why do we need to perform leader election?
Answer: This is because the process starting the election does not know the highest ID process that is alive at that point in time.

Question: If two elections are going on simultaneously, what happens when the first one concludes?
Answer: The second election continues until its completion.

Question: Does election have an ID?
Answer: Yes, so as to facilitate multiple elections occurring simultaneously.

Question: What if the leader crashes even before it is announced?
Answer: You will still announce the chosen process as the leader and it will continue to remain the leader until some process notices that it is no longer alive and initiates another election algorithm.

Question: What if a node with a higher process ID comes up in the middle of an election?
Answer: Leader Election results are stable only if the nodes are stable throughout the election process. If a node suddenly comes up, it may or may not be elected as the leader, but eventually it will be elected when there is another election.

Question: What if the ring is so large that it's not feasible to get a stable result?
Answer: That will be a problem as there is a higher chance of nodes going down or coming up. Some sort of a hierarchy can be introduced in this situation such that there are smaller stable groups and they can elect their own leaders leading to decentralization of the system. However, since this is a P2P system, the stability problem in large rings will remain.

Question: How do you announce the leader? Is it a message broadcast?
Answer: The leader information is circulated along the ring itself by the initiator of the election. It is not a one-to-all broadcast.

14.1.4 Time Complexity

- Bully Algorithm
  - $O(n^2)$ in the worst case (this occurs when the node with the lowest ID initiates the election)
  - $O(n-2)$ in the best case (this occurs when the node with the highest ID that is alive initiates the election)
- Ring Algorithm
  - Takes $2(n-1)$ messages to execute. First, $(n-1)$ messages are sent during the election query, and then again $(n-1)$ messages to announce the election results. It is easy to extend the Ring Algorithm for other metrics like load, etc.

z

14.2 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 is complete, the lock is released. Consider the example below:

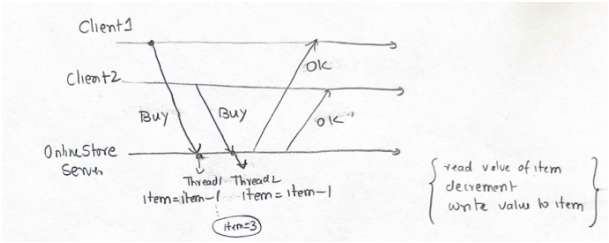


Figure 14.2: Example of a race condition in an online store.

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 a chance of race condition and the 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 the first thread is updating the value (it will read 3) and update it to 2 and save it, which is incorrect. This is an example of a trivial race condition.

14.2.1 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, as shown below in Figure 14.3.

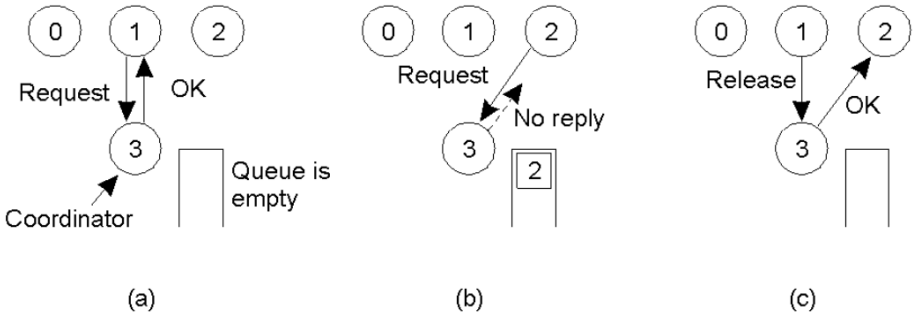


Figure 14.3: Depiction of centralized mutual exclusion algorithm.

Advantages:

- **Fair allocation**: Queue maintains First Come First Served system
- **Simplicity**: Only 3 types of messages are required - *request* (client requests the lock), *grant* (coordinator grants the lock), *release* (client releases the lock)

Issues with handling failure:

- **Coordinator crashes:** When the 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 the 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.
- **Client process crashes while holding the lock:** This is a bigger problem. In such a case, the coordinator is just waiting for the lock to be released while the other process has gone down. We cannot use a timeout here, because client process transactions can take an 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 crashed, it may not always be advisable to forcibly withdraw the lock 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 a centralized algorithm.

Question: Can you check whether the process holding the lock is alive rather than forcing it to release the lock?
Answer: Yes, this is a feasible solution. When the lock holding time expires, the coordinator can just ask the process whether it wants to renew the lock for another time slice or give the lock up. You can grant locks in intervals of time and have the provision to extend that time so long as the process is alive and willing to hold the lock.

Question: What about consistency when the process holding the lock crashes?
Answer: This is an application-level problem. It needs to be handled by the application logic such that all transactions happening within the synchronized block adhere to the desired ACID properties.

Question: If a client process makes a request and does not get a reply from the coordinator, will it assume that the coordinator has not replied or that the reply was lost?
Answer: We assume that network transmissions are handled by TCP which takes care of lost packets and other transmission-related issues. Hence, the client process assumes that the coordinator has not replied yet.

Question: What if the coordinator crashes *while* writing to disk resulting in a mismatch in the data on disk and the process that actually holds the lock?
Answer: Essentially, we need atomicity for the two operations: granting the lock and writing this action to disk. We will cover this topic in greater detail in the Distributed Transactions section of this course.

14.2.2 Decentralized Algorithm

Decentralized algorithms use voting to figure out which lock request should be granted. 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 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, the 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.

Question: If the minority of processes say NO to a lock request, will there be inconsistency?
Answer: Since we are taking a majority vote here, we will assume that the majority of the processes have the correct state of the lock. Normally, there wouldn't be any inconsistency issues. Only inconsistency comes when a process crashes and is not aware of the state of the lock after it restarts. Such a process will tend to give bad replies and this issue is overcome by relying on the majority to give the right replies.

Question: If we want to overcome getting bad replies from processes, can we use techniques like process start time?
Answer: Strictly speaking, the system can force a process to first recover its lock state fully before it can start responding to lock vote requests. However, this is not used in the method currently because we are anyway robust to the bad replies as long as they are in minority.

Question: Is it possible that nobody gets the majority votes?
Answer: If more than half the processes of the system crash, then this will happen as although the minority of processes that are alive have the correct state of the lock but the majority is sending garbage back during voting.

Question: Is there an associated *release* message for every *request* message?
Answer: Yes, once a process is done executing its critical section, it sends a *release* request to all other processes so that they can update their lock state and grant the lock to the next process in the request queue.

Question: In the decentralized algorithm, what are we voting on?
Answer: The vote is simply a *yes* or *no*. Every process responds with *yes* or *yes* based on whether it wants to allow the asking process to acquire the lock or not.

Question: If a process is not using the lock, will it say *yes*?
Answer: In this technique, every process is running the lock manager code as well. So they are aware of the state of the lock and they will grant the lock based on that state.

14.2.3 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.

When any process receives such a request, it either sends a *grant* message (if there are no other requests) or does not reply if it's executing the critical section itself. If there are multiple pending lock requests (including its own, if it needs the lock), it compares the timestamp of the *request* messages and sends the *grant* message to the process that asked for it first.

- Process k enters the critical section as follows:
  - Generate new timestamp $TS_k = TS_{k+1}$
  - Send request(k, TS_k) to all other $n-1$ processes
  - Wait until it receives the *grant* message from *all* other processes
  - Enter the critical section
- Upon receiving a request message, process j
  - Sends reply if no contention
  - If already in the critical section: does not reply, queue request
  - If it wants to enter itself, compare TS_j with TS_k and send *grant* message 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.

Question: How does it deal with failures? In case some process p_j crashes, the requesting process p_i will keep on waiting for the *grant* message from p_j forever without knowing that it has crashed.
Answer: It is a big problem in this system, there are n points of failure. This is one of those cases where a distributed or decentralized system has worse failure resiliency compared to a centralized system.

14.2.4 Token Ring Algorithm

In the *token ring algorithm*, the actual topology is not a ring, but for locking purposes, there is a logical ring and processes only talk to neighboring processes. There is a token that circulates through the ring and any process that has the token is considered to be holding the lock at that time. When the process is done with its transaction, it sends the token over to its neighbor in the ring. The neighbor keeps the token only if it

needs the lock at that moment, otherwise, it passes it on to further. If any node wants to acquire the lock, it cannot ask for it, but it just needs to wait until the token comes to it.

This algorithm derives from the design of an older networking protocol called the Token Ring, which existed as an alternative to Ethernet. In physical networking, only one node can transmit data at a time. If multiple nodes transmit simultaneously, there is a chance of collision leading to garbled data. Ethernet handles this problem by detecting collisions and retransmitting with a back-off policy. In Token Ring, this was handled using locks. Only one machine on the network has the lock at any moment and it transmits at that particular time only.

One problem in this algorithm is the loss of the token. If a process currently holding the token crashes, the token is lost. In the networking algorithm described above, any coordinator node can regenerate the token after a given timeout, however, it is non-trivial to do so here. This is because we cannot put a timeout on the execution time of the critical section, otherwise, it might lead to a scenario where two processes think that they have the token.

Question: Why is a token necessary?
Answer: In the networking analogy, it is ok to use a simple round-robin strategy that allocates a particular time interval (let's say a few milliseconds) to a node to transmit its messages. If it doesn't complete transmission in that interval, it will have to wait for the next interval. However, in the case of distributed locks, if a process is given the lock but doesn't need it, there is no need to wait for those few milliseconds for its turn to be over. If it does so, it will greatly reduce overall performance. If we use the token, the process can simply pass the token to the next node immediately.

Question: Can we add additional functionality for e.g. we can check whether a process is alive and holds the token and is still executing the critical section or it has crashed requiring the token to be regenerated?
Answer: Yes, these enhancements will work but they are not present in the Token Ring algorithm of the networking world. These features need to be added to the algorithm.

Algorithm	Messages per entry/ exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk	2m	starvation
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Figure 14.4: Comparison of Distributed Locking Algorithms.

Distributed Locking is a hard problem to solve. As shown in Figure 14.4, there are advantages and disadvantages to all three of these distributed algorithms.

14.2.5 Chubby Lock Service

This is a distributed lock service developed by Google for internal use. It is designed for coarse-grained locking and uses file system abstraction for locks. Each Chubby cell (a set of 5 machines stores the state of the locks and provides service for up to $10,000$ machines. If you need a lock, you can go to Chubby to get a lock and then use it in your application. It may appear to be a centralized algorithm but there is a set of 5 machines that manages the locks.

It uses basic file system locking (read-write locks) to grant locks to the clients that are requesting them. E.g. if we create a lock named foo, it will create a file named foo and launch a thread that tries to acquire a lock on that file. If it succeeds, the lock is forwarded back to the client process that requested it.

Lecture 15:Distributed Snapshot

15.1 Overview

This lecture covers the following topics:

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

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

15.2 Transactions

Transactions provide a higher mechanism for atomicity of processing in distributed systems. *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.

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. Let's further assume accounts A, B, C has \$100, \$200, \$300 respectively initially. 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 from client 1 and client 2. 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.

Clients are executing parallel queries, so can't assume anything about the order in which the operations take place. 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.

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.

One way to achieve this would be to use a lock on the entire database, so only one of client 1 or 2 can access the database at a time. The issue with that is it would make all the operations on the database sequential (i.e. only one client can write to it at a time) but you may have multiple clients sending requests at the same time, so the performance would degrade significantly. So what we want is for the database to physically execute concurrently, while logically it seems like it is executing sequentially, which is achieved through transactions. Transactions essentially allow you to take locks on a set of operations.

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

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.

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.

Question: If there is a rollback in a transaction, at what point can it happen?

Answer: In the example above, client A has four instructions which are together enclosed in a transaction. You can decide to abort a transaction at any point during the execution, in which case the amounts will revert to the original value. But once you have committed the transaction then there is no going back and all the changes become visible.

Question: How are multiple requests (i.e. multiple clients) executing together?
Answer: This will be achieved by something called concurrency control, which we will get to. In brief, we have to implement finer grain locks- not locks on the whole database but locks on individual records.

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.

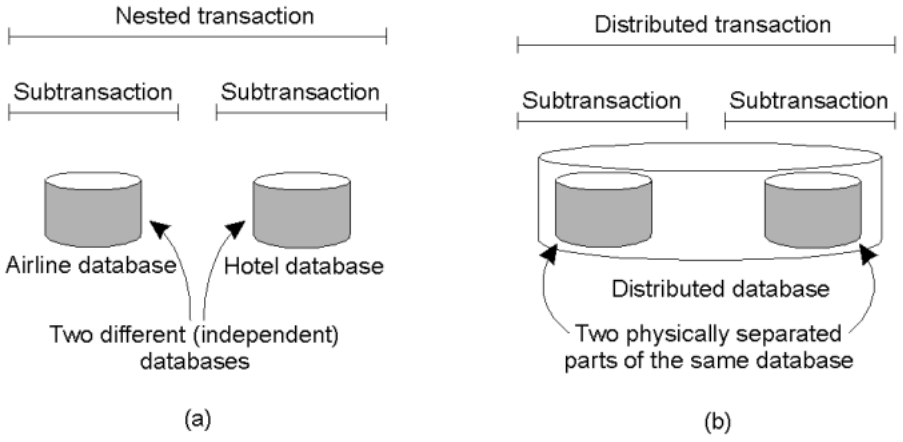


Figure 15.3: Nested transactions and distributed transactions.

15.2.3 Distributed Transactions

Two concepts: nested and distributed transactions

a) *Nested Transaction*

Here one transaction is nested inside another. That is, within the BEGIN and END block that denotes one transaction there is another BEGIN and END block.

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*. 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 a database that is distributed (i.e. not on one machine). 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.

Transactions are not database specific, even though we talk about them in that context. We can have transactions in a distributed web app if we want.

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

We will see two ways to implement distributed transactions- private work spaces and write-ahead logs. Both these methods work for both single transaction systems as well as distributed transaction systems.

Private Workspace

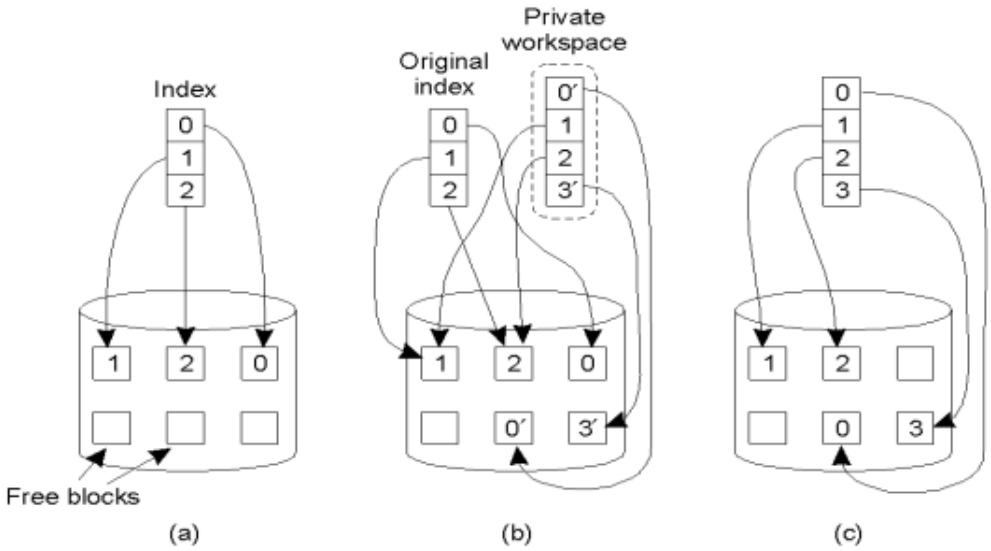


Figure 15.4: Private workspaces.

- 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 the above figure, 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*. Downside: making a copy of a large DB is not cheap; a lot of work if there are small changes. So, we need very efficient way for making copies. A virtual copy will allow for efficient copies – use *copy on write*. A private workspace is a copy of the entire DB.

Question: At any given point in time is there only one transaction being processed?
Answer: That is not the case, we want an arbitrary number of transactions to execute in parallel. Each transaction will have its own private workspace- for N transactions we will have N workspaces (and the original database) where they make their changes.

Question: If two transactions make copies to the same block at the same time what do you do?
Answer: That is called the write-write conflict. Either you have to abort both and restart them or let one of them succeed. In private workspaces, aborting the workspace is as simple as just discarding the private workspace index copy that was created. In a transactional system, whenever there are conflicts the transaction will abort.

Question: Will the index be stored on a single machine or distributed machines?
Answer: At this point we are just looking at a logical view of what is happening. The notion of private workspace works well if you have a single machine and database or multiple machines and databases. If you have many machines then you have private workspaces that span multiple disks, but the concepts remain the same- you use copy and write, with efficient snapshots, and commit if there are no conflicts or discard if there are.

Question: Is this used in real systems, since there are accounts being changed that will cause issues if they are discarded?
Answer: We will talk about concurrency control, which is a process by which you let multiple transactions execute in parallel and yet get safe results. A version of that is called optimistic concurrency control where you don't use locks, allow transactions to make changes, and at the point where they are about to commit check if two transactions have overwritten each other. If they have we declare a write-write conflict and abort. Most transactions are not modifying the same piece of data, so with high probability will successfully commit. In the case where you do have multiple transactions frequently modifying the same piece of data then optimistic concurrency control will lead to many abortions, so we will instead favor pessimistic concurrency control.

Question: When do you merge the change? Can you merge it while another transaction is ongoing
Answer: Yes, there can be an arbitrary number of transactions ongoing. The only thing we need to know

is when we started executing did someone else also modify the same data items that you did, in which case you abort.

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 live database instead of a copy. We instead keep a transaction log in separate file (called a write-ahead log) to note the changes the transaction is making to the database. Here committing is trivial since the changes are already executed on the live database, but aborts are harder because we will have to undo the changes by scanning the write-ahead log and undoing each operation listed to restore the database to where it was before the transaction. The undo process is called a *rollback*.

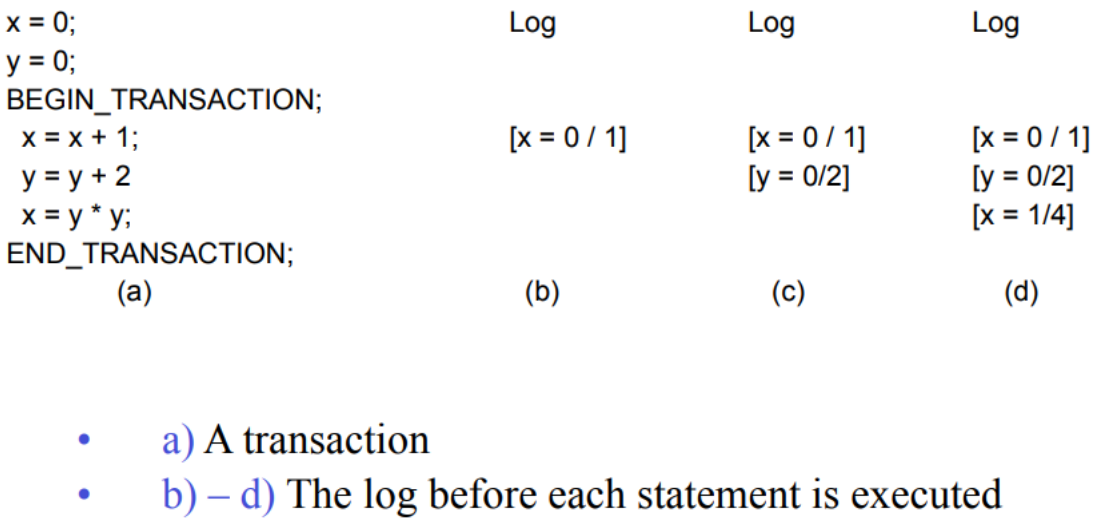


Figure 15.5: Distributed transactions

The above figure 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, and entry with the original and updated values 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, another new entry 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 undo the changes, the log is traversed backwards reverting each operation by replacing current value with the previous value.

Question: How does the transaction know if it's aborted?
Answer: Two ways to abort: explicit statements that the user wrote, or locking as we will see. The lock manager will abort if needed.

Question: If we're already making changes to the database and another transaction comes and modifies the data how will it be handled?
Answer: When it is time to commit, we have to see if anyone else has modified the changes we did, in which

case we flag a conflict and undo the changes. The transaction system can only do one of two things- either as you make the changes you take locks from x and y so that no other transaction can modify it (pessimistic case), or if we don't hold locks while making changes, we check at commit time if anyone else made changes, and if so we abort the same thing (optimistic case).

Question: Is the log per transaction or is there one log shared by all?
Answer: The transaction log is a single file. Multiple transactions are going to write to that file, what isn't shown here is that each of these entries has to have a transaction ID associated with it, so that when we are scanning the log back we just look for that ID in order to revert changes.

Question: When you roll back, do you have to scrape out the IDs?
Answer: You don't have to change the log. The log will have aborted transaction, you just have to undo what is in the database. You don't have to delete from the log.

Question: What is the case where the optimistic approach is better?
Answer: The idea behind optimistic approach is that for a large database with millions of records, and any given transaction will only make changes to a small part of the database. Thus, even if there are many transactions, if each work on different parts of the database they will rarely conflict.

Question: If C is about to commit and it sees that B has made changes, so you go to abort C, and when B is about to commit you see it has caused problems with A, how does the abort take place?
Answer: Here you have a case called the cascading rollback, where each abort causes a problem with another transaction due to a write-write conflict, so both have to termintated, which conflicts with another one and so on. That is possible, and is a bad case because multiple transactions all get rolled back at the same time. The only solution here, to avoid many conflicts, is to take locks.

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

The goal of concurrency control is to allow several transactions to execute in parallel. We want to avoid each transaction taking a lock on the entire database because it would lead to poor performance if many users want to execute transactions at the same time. In other words, we don't want sequential execution (wait times would balloon during heavy loads), which is what *global locking* will give us and won't allow concurrency. Instead, we want all the transactions to execute in parallel while still achieving consistency. If all the transactions commit successfully, the final result should appear as if they executed sequentially.

Concurrency can be implemented in a layered fashion as shown in the prior figure. The transaction manager implements the private-workspace model or write-ahead log model. The scheduler implements locking and releasing the data (in this case, we are taking locks of records of the database rather than the entire database). 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 the next figure. Data is now split across multiple machines. A scheduler is on each machine and needs to handle distributed

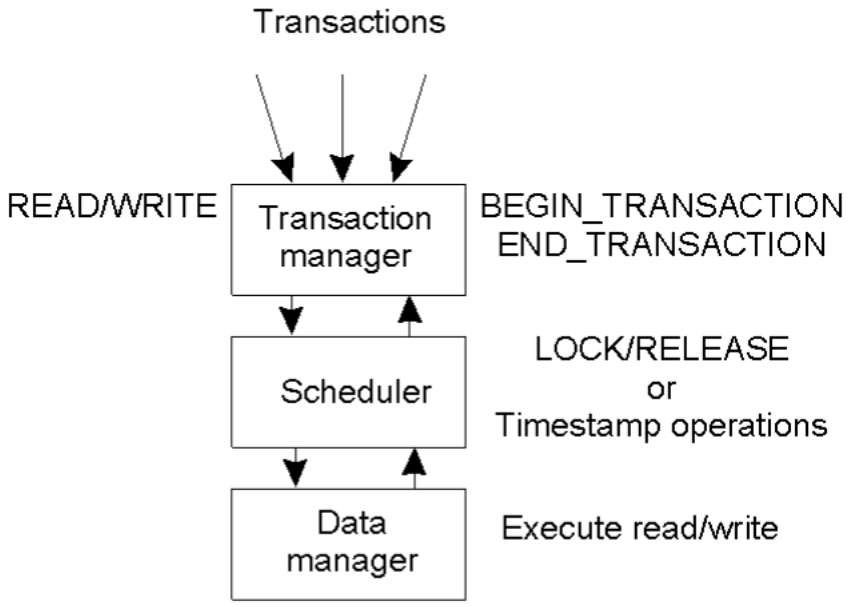


Figure 15.6: General organization of managers for handling transactions.

locking now. Beyond that everything is same. Question is what should be in the scheduler for us to get concurrency? We will look at that next.

15.3.1 Serializability

This is the key property imposed on the end result of a transaction: *idea is to properly schedule conflicting operations*. The end result of concurrent transactions should be to look as if the transactions were executed serially. The next figure 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 the six operations associated with the three transactions (two in each transaction) execute in parallel, we can obtain any arbitrary interleaving of the six operations. We then check if there is a sequential execution of transactions that could have produced such an interleaving, in which case it is considered valid. The last transaction will set the final state of 'x'.

In the above example, Schedule 1 is valid because the output is same as if the transactions are executed in the a,b,c serial order. In Schedule 2, we see some interleaving of instructions- x=0 is executed twice before x=1 and x=2. Here we check the end state, which is that x = 3, and check if there is some sequential order that would lead to the same end state. Since there is such an order (the same a,b,c order as Schedule 1), we say that Schedule 2 is also legal. Schedule 3 is illegal, however, because the end state interleaves x as value 5, and there is no sequential execution that could achieve the same end state.

Question: Is the scheduler actually going to simulate this entire process?
Answer: The scheduler is not going to do that. This is just a concept called serializability- if you have operations executing concurrently, the output of these operations should be as if they were executed in some sequential order. If you want to implement serializability, it is not the scheduler who should figure this out, you would rather have some protocol do this automatically.

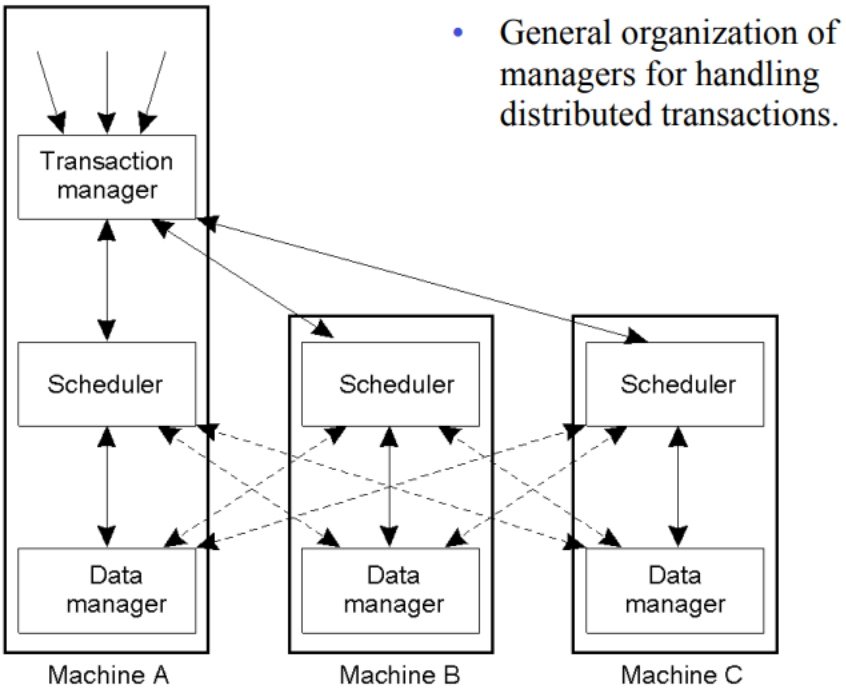


Figure 15.7: General organization of managers for handling distributed transactions.

```
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.

Interleaving could result in two kinds of conflicts: read-write conflicts and write-write conflicts. Read-write conflicts occurs when the transaction reads, performs a write based on the value of the read data, but sees that the data has been updated since the read took place. 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 the *optimistic concurrency control* technique, there are no locks or lock manager. 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 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.

While there are some distributed services that use this approach, it is not widely used, especially by commercial databases because of high abort rates at high loads.

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)

Pessimistic concurrency control requires the use of locks. One widely used protocol for this is called two-phase Locking (2PL), which 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.

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 the above figure. 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.

A simpler approach is to completely avoid shrinking phase. As shown in the last figure, Strict Two-Phase Locking grabs locks and releases all of the locks at the same 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

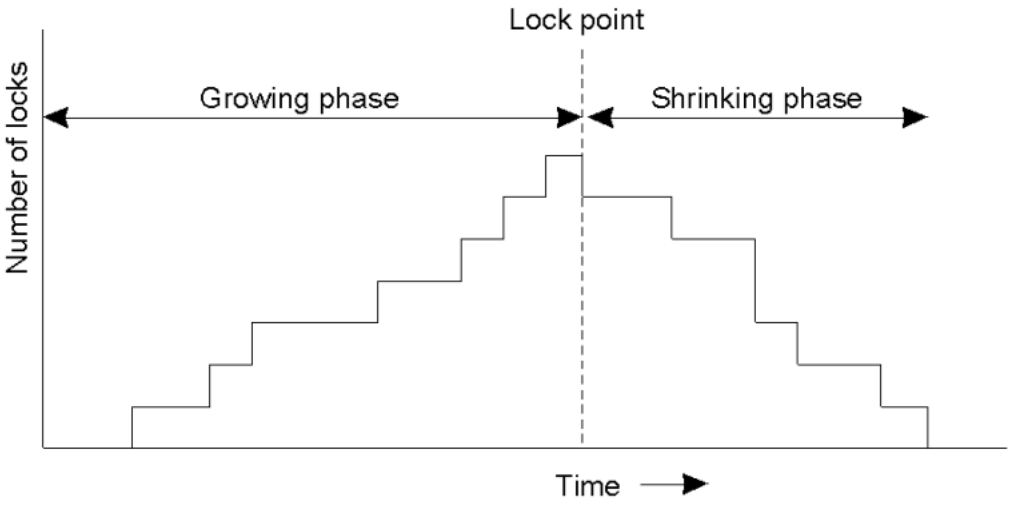


Figure 15.9: Two-Phase Locking

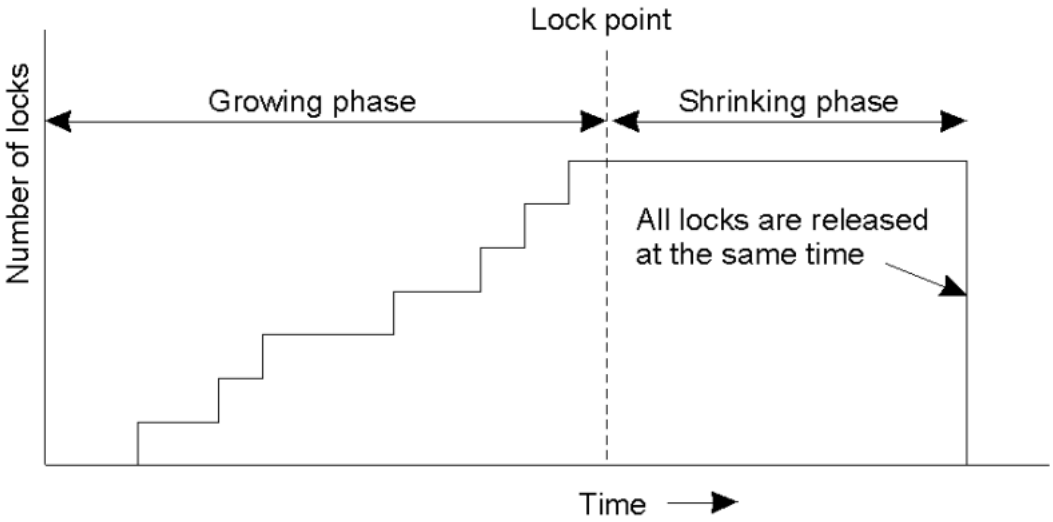


Figure 15.10: Strict Two-Phase Locking

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 (not locks), in particular Lamport's clock (logical clock instead of physical 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.

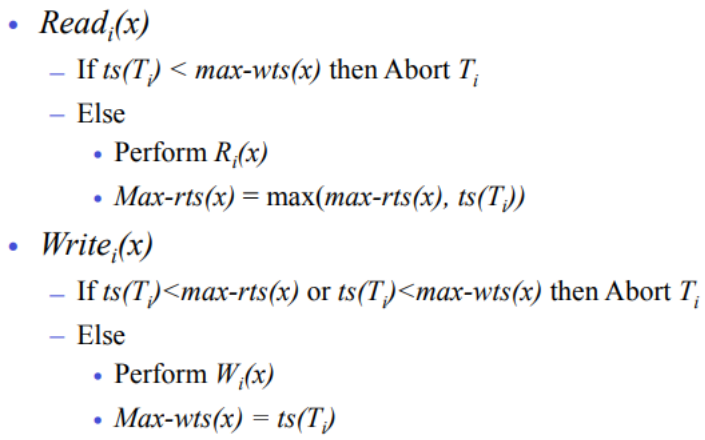


Figure 15.11: Read-writes using timestamps

Conflicts are handled using both these timestamps as shown in the Figure 16.13. 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 16.13?
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.

Lecture 16:Token Ring Algorithm

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 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.
- **Consistency of Replicas**
  If one copy is modified, others become inconsistent.

16.1.2 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.

16.2 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.

Question: Are single database available as shown in slides?
Answer: Single databases are not available, if single databases goes down.

16.2.2 NoSQL Systems and CAP

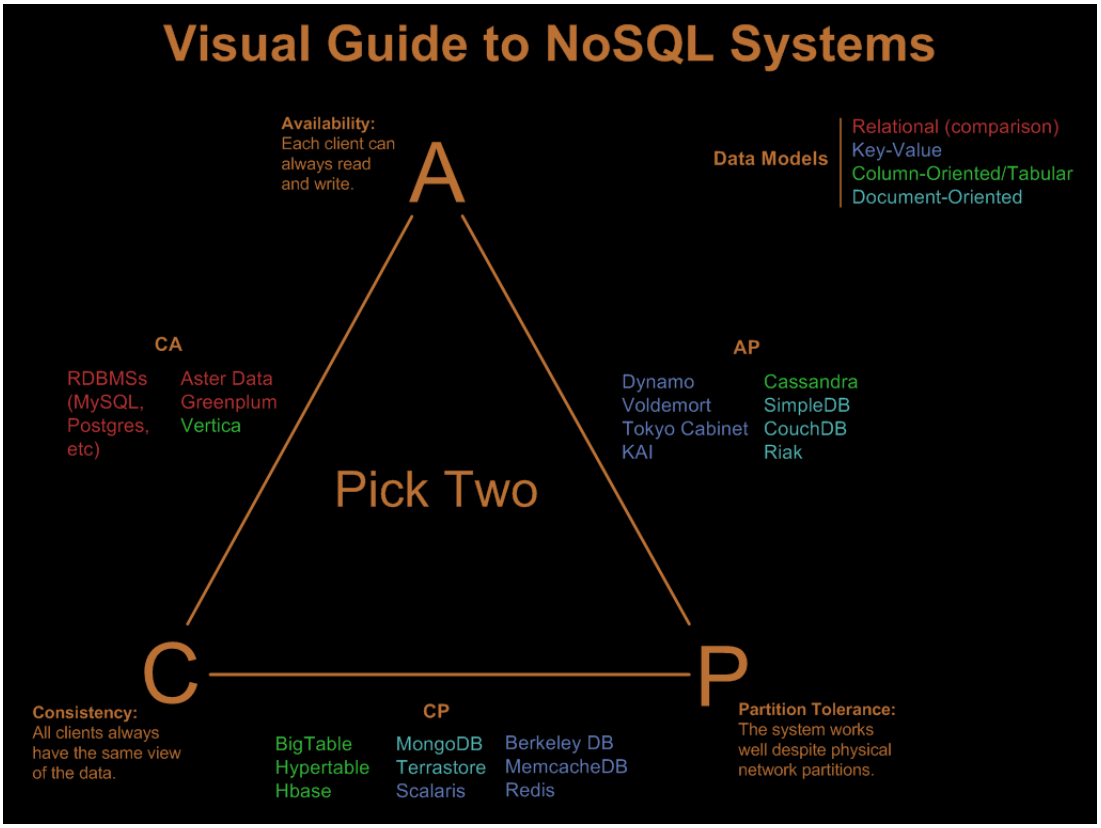


Figure 16.2: CAP in Database Systems

NoSQL systems, as the name suggests, don't use a SQL database. Figure 17.2 shows some database systems and which properties they hold. Consistencies in the presence of network partitions are problematic.

Question: (previous year) 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: (previous year) 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: (previous year) 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: (previous year) In Figure 17.2, 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

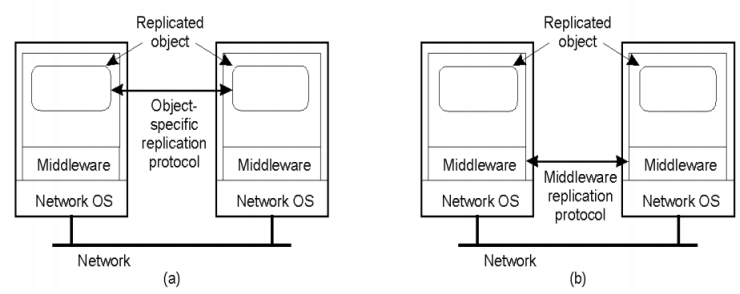


Figure 16.3: Two types of replication. (a) The application does the replication and handles consistency. (b) The middleware does the replication and handles consistency.

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

Question: (previous year) What do you mean by "Simplifies application development but makes object-specific solutions harder"?
Answer: The developer doesn't need to deal with the replication logic for the application, but the developer will have to stick to the replication and consistency scheme provided by the middleware. There are many such schemes as we will see ahead. It reduces flexibility for the developer.

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. There 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.

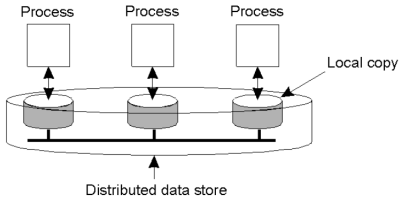


Figure 16.3: Data-centric consistency models.

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.

Question: When we say CAP theorem, do we mean strict consistency, or looser model of consistency?
,

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.

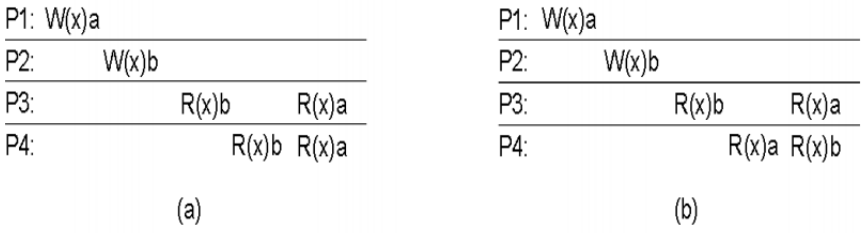


Figure 16.4: Sequential Consistency

In Figure 17.4, 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 17.4 (a) The processes agree on the order that $P2$ wrote before $P1$ and both $P3$ and $P4$ read in that order. Figure 17.4 (b), $P3$ and $P4$ see in different orders and that is not allowed.

One reason for operations to have different orders could be because the individual servers are located geographically apart. The propagation delays can cause the ordering to change. In sequential consistency we allow any ordering as long as all processes agree to that ordering.

Question: How do processes agree or not?
Answer: There are consistency protocols which implement this.

Question: (previous year) 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 $P2$. 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.

Question: (previous year) Would it be expensive to implement the order?
Answer: The implementation will always have an overhead/cost to it which we will see later. Right now we are just discussing the concepts or types of consistencies which we can implement depending on our use cases.

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 17.5 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 17.6.

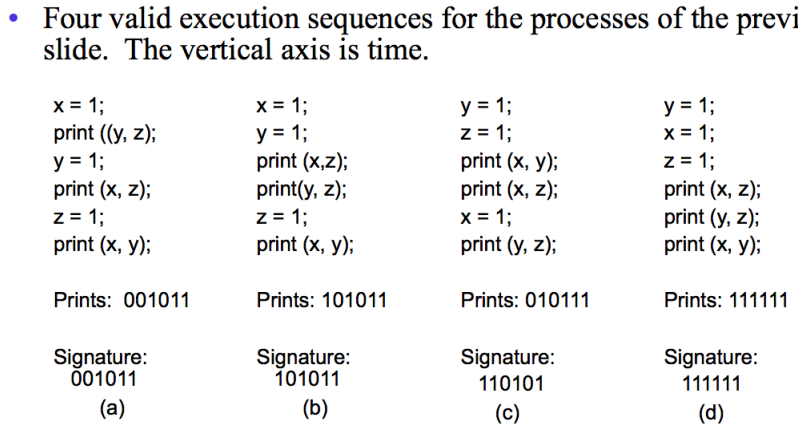


Figure 16.6: Valid interleaving for Figure 17.5 satisfying the property of linearizability.

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 17.7 (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 17.7 (b)). Causal consistency is weaker than linearizability as the latter fixes an order.

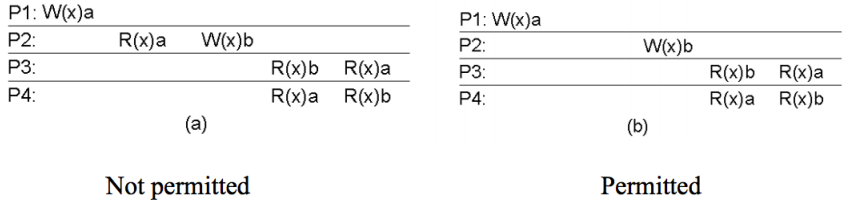


Figure 16.7: Causal Consistency

Question: Can order for writes as seen by processes be different?
Answer : Order only matters if two writes are causally related. For concurrent writes, order doesn't matter.

Question: In what scenarios, sequential consistency gives you better properties than causal consistency
Answer : Social media post, for example, if two users post comments and users see it in different order, they don't know who commented first. Causal consistency would be more limiting in it.

Question: (previous year) In Figure 17.7 (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: (previous year) Could you give an example for what is allowed in sequential consistency but not in linearizability (Figure 17.7 (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: (previous year) In lineraizability is $R(x)a$ $R(x)b$ and $R(x)a$ $R(x)b$ allowed (Figure 17.7 (b)?
Answer: Yes, in fact that is required. You have to see a followed by b if there is a happen-before relation.

Question: (previous year) What is a causal relationship? **Answer:** There is a causal relationship between 2 processes if they communicate. If one process writes to a shared variable and another reads from it, the variable becomes a shared buffer. This establishes a causal relationship between the 2 processes. Similarly we can say about message passing between processes.

Question: (previous year) If $P4$ was $R(x)b$ $R(x)a$ in Figure 17.7, will it be permitted? **Answer:** No. It will not be permitted. The $R(x)a$ in $P2$ causally relates the writes in $P1$ and $P2$. Since the $W(x)b$ comes after already reading the value written by $P1$ as a, only permitted order is $R(x)b$ after $R(x)a$. If there was no $R(x)a$ in $P2$, any order will be permitted for the reads in $P3$ and $P4$.

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 neces-

sarily 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: Does the system enforce these rules?
Answer: Most systems have very poor consistency guarantees, practical example being password being changed by user on one device is not reflected for a long time in the other device.

Question: (previous year) 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 examples 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: (previous year) 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.

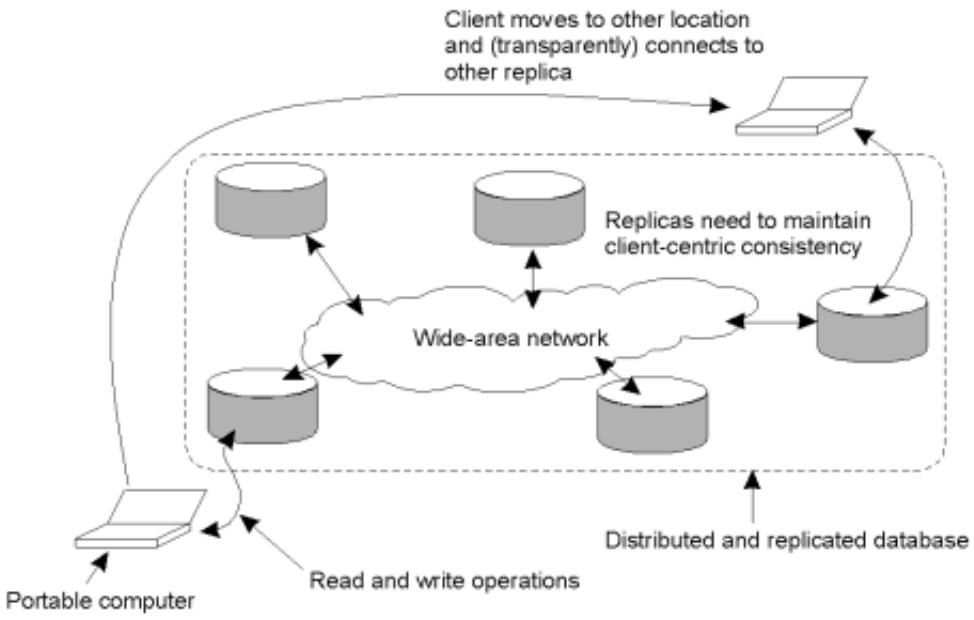


Figure 16.8: Eventual Consistency

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: (previous year) 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: What is k?
Answer: 1/k is the probability of stopping, k is a tunable parameters that tells you how many nodes you will contact before you stop.

Question: How does rumour mongering guarantee eventual consistency?
Answer: Rumour mongering doesn't guarantee eventual consistency, but so would Anti-entropy as it might take infinite time to spread the update. So eventual consistency is so loose that it actually doesn't guarantee consistency.

Question: (previous year) 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: (previous year) 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: (previous year) If a node is trying to pull, how does it know what is hot and what is cold?
Answer:

1. Rumor Mongering: It is push based with a probabilistic backoff based on if the receiver already knows about the change or not.
2. Anti-entropy: If a node tries to pull from another node who hasn't seen the change, the node will also not see the change.

This is why it is better to have a combination of push and pull methods so that nodes which do not receive the changes in a push can pull from those who have. This is a better approach for eventual consistency.

Question: (previous year) What happens in an only pull based model?
Answer: A node will tell another node that it has not seen any changes since 12 PM. If the other node has any changes after that, it will send those to the first node.

Paiwise Exchange is when you do both push and pull. For example, in a FitBit that is connected to your smartphone, it will periodically connect to the phone and push data like health metrics. At the same time, it will also pull data from phone like weather info, etc.

Question: (previous year) 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.

Question: What happens if you relax the consistency in CAP theorem?
Answer: If you give up consistency, you get availability and partition tolerance, which means that even if the network is down, you have some replicas on the left half, and some on the right, and you get to access them. Which is better depends on the context of application, and there is no single right answer.

Lecture 17:Concurrency Control

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. Since all nodes have a copy of the file, local reads are permitted.

   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, is there a primary node for each data item?
Answer: Yes, there is a primary node for every data item. Each node can be chosen as primary for a subset of data items.

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 clients 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.

Question: How do you handle concurrent writes in Local-Write protocols?
Answer: When there are concurrent writes, there is no need to move the primary as you can not determine the primary in a concurrent write situation. In this case it will be a remote write.

Question: Are reads blocked in Synchronous Replication?
Answer: Reads are local, hence they're not blocked.

Question: Can you improve consistency in asynchronous replication by sending additional messages?
Answer: The issue originally with asynchronous replication is in the case of crash failures of the local machines, when the write can not be propagated to the other replicas. Thus, sending additional messages is not fruitful in these cases.

Question: Can you do asynchronous replication without waiting for any of the operations to be successful?
Answer: Yes, you can do that by sending a message when OS received the write request.

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.

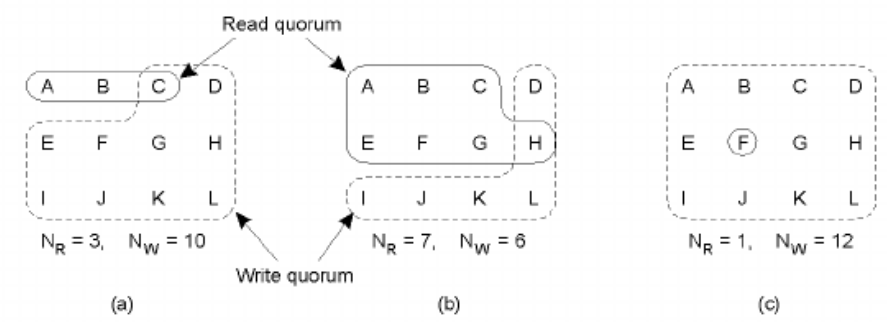


Figure 17.1: Different settings of N_R and N_W.

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 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: How do write quorum nodes agree on a update?
Answer: The main focus of the agreement is to ensure that all nodes complete the writes and that all of them are using the same version number. Some of the ways of agreeing on a version is to use the current timestamp for the update version. The version could also be a simple integer which is incremented for an update.

Question: In examples (a) and (b) in Fig 17.1 where the data is not replicated to all nodes, will there ever be a case in which the read quorum never agree.
Answer: If the rules from equations 17.1 and 17.2 are followed, then your are guaranteed that the read quorum will always agree. However, if arbitrary values are chosen for N_R and N_W that don't follow these rules, the nodes might agree on reads and writes but the results may not be correct.

Question: Why can't you just return the file with the highest version number?
Answer: It can be done however, this is a voting based protocol. So we need to ensure that the files match at multiple replicas to ensure the correctness of the file returned. Thus, the protocol can be simplified by adding version numbers, but it is not necessary and the protocol will work regardless.

Question: Why is returning an older version allowed?
Answer: While a new write is in progress, the most consistent version of the file is the older file, hence it is allowed.

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?
- 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. Unlike centralized applications, where a program crash results in a complete application stoppage, a well-designed distributed system can continue to function even when one or more machines fail. Failures themselves could be hardware crashes or software bugs (which exist because most software systems are shipped when they are "good enough")

Fault tolerance 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.

If a system has n nodes, and the probability that single one fails is p, then the probability that there is a failure in the system is given by:

$$p(f) = 1 - p^n$$

As n grows, this number probability converges to 1. In other words, there will almost always be a failed node in a large enough distributed system.

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.

Lecture 18:Implementing Consistency Models

18.1 Overview

In previous lecture, we have started Fault tolerance, in this lecture we will continue it and cover following topics

- Agreement in presence of faults
- Reliable Communication
- Distributed Commit

18.2 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. Byzantine faults are much more difficult to deal with.

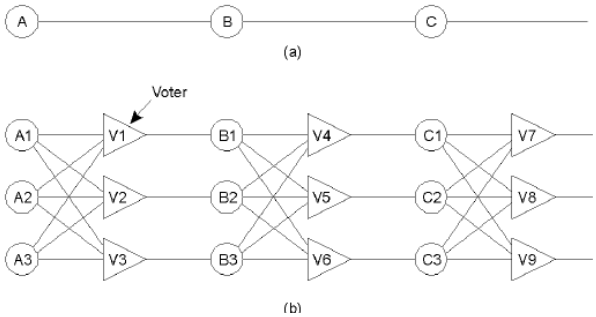


Figure 18.1: Failure masking by redundancy

Question: Is Figure 18.1 for Byzantine or not?
Answer: You can do Byzantine with it. Crashes are easy. Crashes can tolerate two faults, but you can do Byzantine as well.

18.2.1 Byzantine Faults

Let's explore two situations where the aim is to reach to an agreement on a one-bit message, Scenario-1: Two perfect process with faulty network (Two Army Problem) Scenario-2: faulty processes with perfect network (Byzantine generals problem)

Two Army Problem

In this problem, two armies in separate camps must agree on whether to attack a fort for the attack to succeed. They communicate via messengers, each sending a vote message ("attack" or "retreat") and waiting for an acknowledgment ("ack") from the other. However, enemy interference leads to unreliable communication, leaving the army uncertain if their messages or acknowledgments were received.

For example, suppose the first general sends an "attack" message. The second general receives the message and sends an "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, which leads to an infinite loop.
Summary : Two perfect processes can never reach an agreement in the presence of an unreliable network.
Question: If the network is faulty, can we not use cryptography?
Answer: We can use cryptography but we still cannot deal with an unreliable network that completely drops messages.
Question:In TCP, 2 ACKs are adequate, in two army problem why two acks are not sufficient ?
Answer: Firstly, TCP uses 1 ACK, secondly in TCP we are not trying to reach an agreement to coordinate something. To know if a message is delivered or not an ACK is sufficient, which is not the same case when we are trying to reach an agreement.
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.
Question: What does it mean to see if the network is reliable?
Answer: Messages are delivered and delivered correctly. For example, TCP does that for us. But, we don't know if the generals can be trusted or not.

Byzantine Generals Problem

In this problem N generals are trying to reach to an agreement with a perfect channel but M out of N generals are traitors.
Problem 1: Here M= 1 and N = 4. A recursive solution to the problem is provided in 18.2 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 decide on one correct value or spot if something's wrong, like a Byzantine failure/traitor.

Question: Does Figure 18.2 fail if the number of traitors outnumber the other generals? **Answer**: Yes.
Question: In problem 1, what exactly we do in round 2?
Answer: Generals broadcast all the information they have from every other general in the previous round.
Question: What happens if a fault process send same incorrect message in round 2?
Answer: If that is the case, it is acting like a normal process and then it won't be identified or isolated. However if, instead of broadcasting their strengths, they perform an addition task, such as computing and returning the value of 2+2, any incorrect answer would be detected in this scenario.

problem 2 : Here M= 1 and N = 3. We do same steps as problem 1 in 18.3

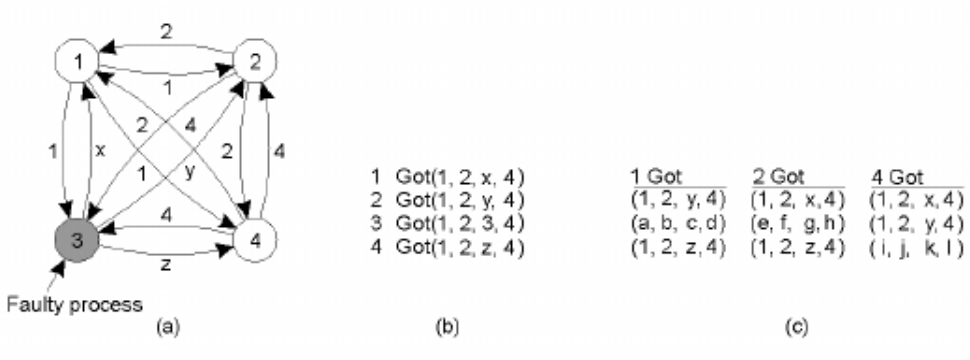


Figure 18.2: Solution to the Byzantine general's problem, 1 traitor out 4 generals over a reliable communication network. a)The generals announce their troop strengths b)The vectors after each general assembles after round 1 c)The vectors that each general receives after round 2.

Figure 18.3: byzantine general's problem-2, 1 traitor out 3 generals over a reliable communication network.

In this case we can detect the fault but cannot isolate it, in order to isolate the faulty process we need one more process which functions correctly.

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. Therefore, agreement is possible only if $2k+1$ processes function correctly out of $3k+1$ total processes.

18.2.2 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 Reliable Communication

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 18.4 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

Broadcast is sending a message to all nodes in a network. Multicast is sending to a subset of all nodes.

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.
- 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**.

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.

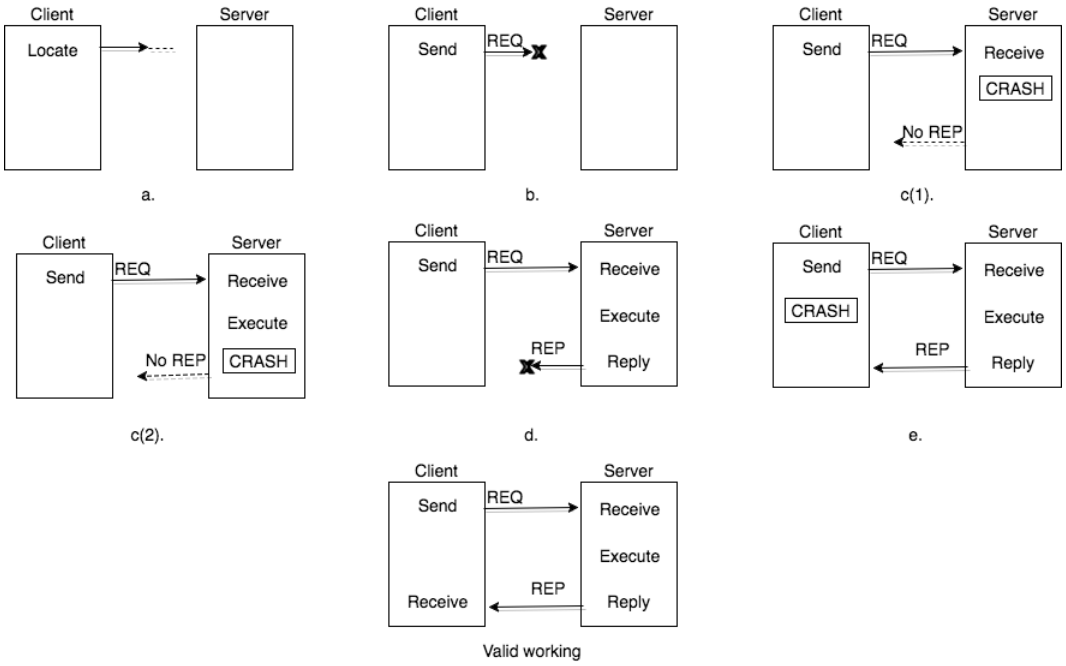


Figure 18.4: Types of failures in the one-to-one scenario. (a) Client unable to locate server. (b) Lost request messages. (c) Server crashes after receiving request. (d) Lost reply messages. (e) Client crashes after sending request.

- 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.
- Much more scalable than ACK-based schemes
- Effective only in networks with occasional drop-offs and is not suitable for highly lossy networks.

Question Are messages not queued at each receiver and delivered in the same sequence? If a message is missed, can't we request it from one of the receivers?
Ans. Yes, we maintain a buffer to order and deliver messages sequentially. In case of a missed message, we send a NACK to both the sender and a subset of nearby receivers.
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.
Question Is a NACK-based part of TCP protocol or part of the higher level application?

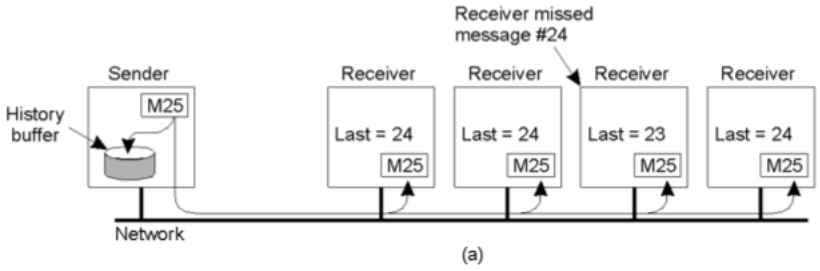


Figure 18.5: Here, all the receivers have their last packet received as #24 except receiver 3 which missed packet #24. Hence, it's last packet is #23. As soon as it receives packet #25, it knows it missed the packet #24.

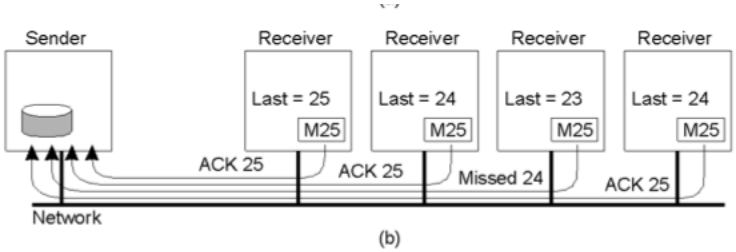


Figure 18.6: Each receiver now sends an acknowledgment ACK either in form of received packet #25 or missed packet #24. As we can see for a single packet, sender receives 'n' ACKs

Ans. You can ask that question of multicast itself. There are two versions of multicast. IP multicast where an IP address is a group address, and sending to that group IP send to the entire group. That is at the network level. You can also implement this at the application level. The NACK based scheme is mostly done at an application level.

Question Can you implement it on top of UDP

Ans. Yes. Application level multicast is implemented internally effectively as n unicast messages. Network level is just one socket, and it goes to n receivers.

Question Is it like a pub sub architecture?

Ans. Not exactly. This is more level than publish and subscribe. This is an abstraction one-to-one communication. You can build publish subscribe on top of this.

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

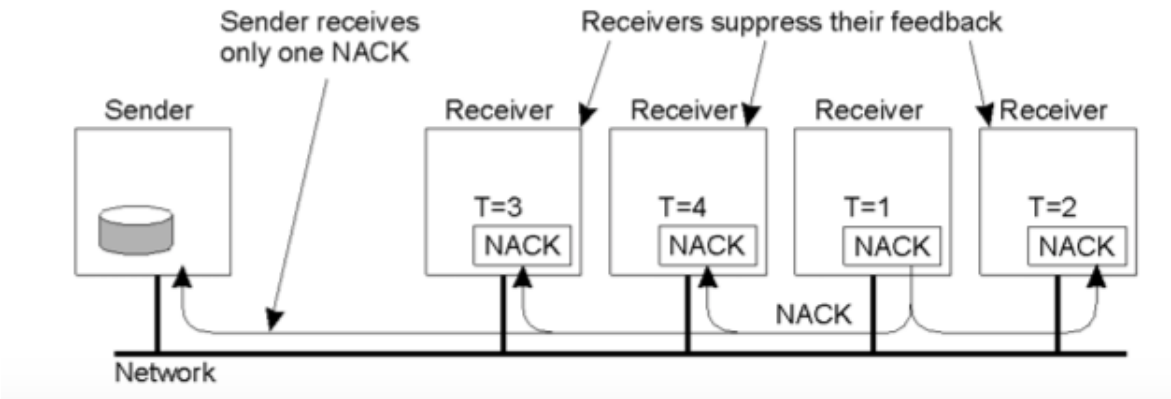


Figure 18.7: Each receiver now suppresses their ACK feedback. Only receiver 3 sends a NACK to other receivers and the sender.

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.

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.

Question: How do we know which processes are up and which are crashed?
Answer: To detect which nodes are up and which are crashed, we can implement several procedures like

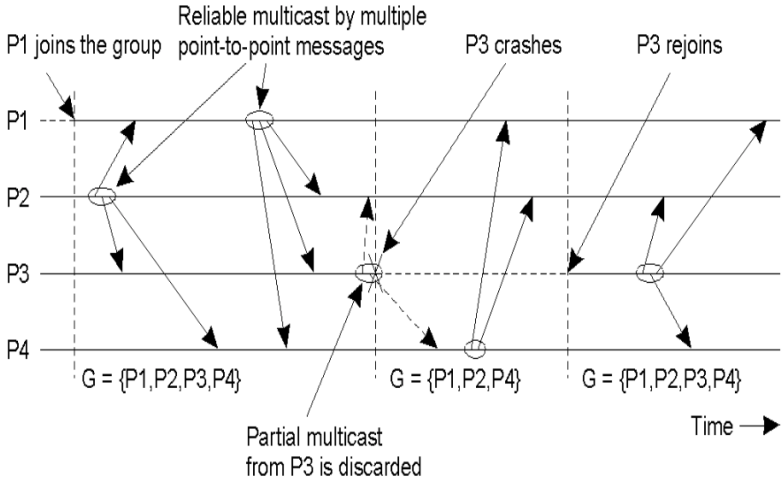


Figure 18.8: Initially all process are up and are part of a group {P1,P2,P3,P4}. All the messages are being reliable multicasted to each of the processes. At dotted line2, P3 crashes while sending a message. From this point onwards, the group {P1,P2,P3,P4} will not maintain the 'all' property of atomic 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.

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 18.9:

heartbeat messages to know the status of nodes. These are crash faults not byzantine, so they are easy to track.

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 18.10

- 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.

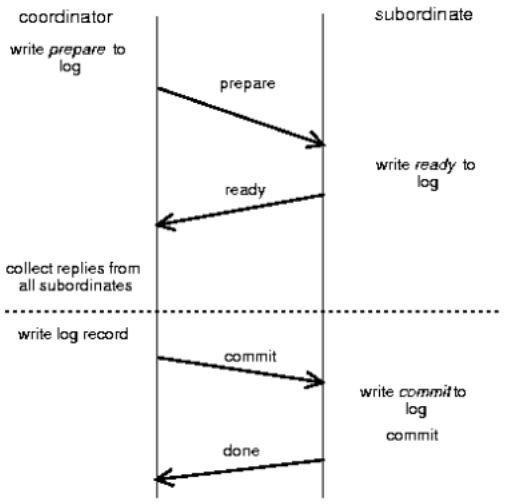


Figure 18.10: Steps showing a successful global commit using 2PC approach

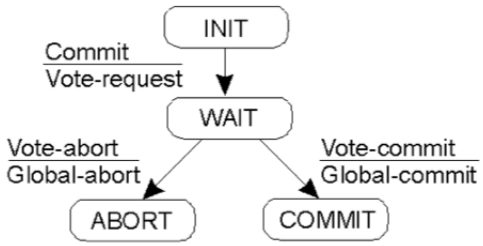


Figure 18.11: **2PC: Coordinator's state transition.** From INIT state, the coordinator asks all processes to vote and goes into WAIT state If any one process votes abort, the coordinator goes to ABORT state and issues global-abort. If all processes vote commit, coordinator goes in COMMIT and issues a global-commit.

Question 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.

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

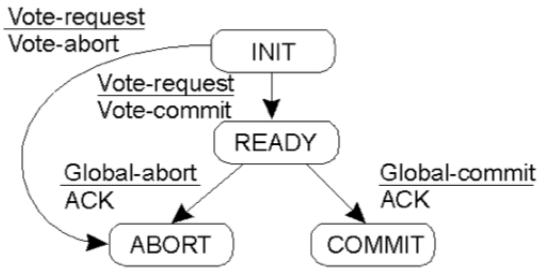


Figure 18.12: **2PC: Subordinate process's state transition.** A process may vote abort and go directly into ABORT state. This is because this single abort would lead to global-abort. A process may vote commit and go into READY state. On being READY and receiving an abort, the process goes into ABORT state. On being READY and receiving a commit, the process goes into COMMIT state.

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.

- 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: Is it feasible to discard coordinator and elect a new coordinator in case of deadlock ?
Answer: we can do that in subsequent process but it is still a deadlock for current process.

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.

Question: what happens if a node crashes after it works to commit?
Answer: Process upon restarting/re-initialisation needs to check it's last state and make a decision.E.g: If it's in init state, that means that it did not vote for the operation otherwise it would have been in the ready state, in this case, it's best to abort and inform coordinator about the decision so it can make progress.

If the process was re-initialised with a "ready" last state, then it means that it voted before crashing but doesn't know the result, so it needs to check other process queues for that operation and figure out the decision.

Lecture 19:Fault Tolerance

Lecture Overview

- Part 1: Consensus
- Part 2: PAXOS
- Part 3 : RAFT

19.1 Consensus

Definition: get a group of processes to agree on something such as database replication etc. Consensus also means getting the set of processes to agree when some of the processes fail via atomic. More formally, we want to achieve reliability in presence of faulty processes:

- Requires processes to agree on data value needed for computation.
- **Examples**: The operation could be anything - whether to commit a transaction, agree on identity of a leader, atomic multicasts, 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 crypto-currency.

19.1.1 Properties of a Consensus Protocol

- **Agreement**: Every correct process *agrees* on the same value. (fundamental to consensus)
- **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.

Q: Difference between agreement and termination A safety and liveness question, Agreement can agree on null value, it's a safety property, and termination means we make agreement on actual value, so it's a liveness property, we need both for the system to go on.

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 failures: 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.
- Either all commit or no one commit.

Three Phase Commit

- It can handle coordinator failures.
- But network failures are still a problem.
- Add an additional stage of pre-commit compared to 2PC when the coordinator receive the commit, and if the coordinator says commit again, all the nodes move to commit stage.

3PC never get confused for abort and commit because of the existence of pre-commit stage. 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.**

Replication for Fault Tolerance Technique 1: split all incoming requests among replicas. (If one replica fails, other replicas take over its load) Technique 2: send each request to all replicas. use 2PC, 3PC, Paxos, a replica can produce wrong results

19.2 Paxos: Fault-tolerant agreement

Paxos lets nodes agree on the same value despite node failures, network failures and network delays. but cannot deal with Byzantine Fault **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.

Question: what method would you want to use this as opposed to other approaches (2PC/3PC) ? *Answer*: This can be used in case of failures in the node as well as multiple leader failures. Since it is a quoram based protocol, it allows leader election even if some processes fail as long as majority of them are up.

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.

Question: Are we assuming the number of nodes is fixed? Can new nodes join? *Answer*: It is not sure new nodes can join since we are assuming because as mentioned previously here that if you have a steady stream of failures and recovery liveness is not guaranteed so we can have failures but then if new nodes are joining and they suddenly start saying something that they're in participant participate that's a problem then that round will fail the nodes can join but that round will failure to return. It may also cause problems in majority voting.

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.

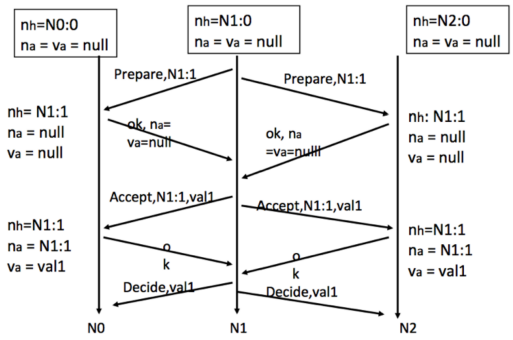


Figure 19.1: Example of Paxos with 3 servers

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.

Q: How can there be two source of truth on network partition: In Network partition, the majority can make decision, but the minority cannot make a decision since it cannot get majority of nodes to agree with.

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.

Note : All the metadata such as who was the leader node, term number etc along with the log vaule needs to match for a logs between process and majority to be considered equivalent.

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.

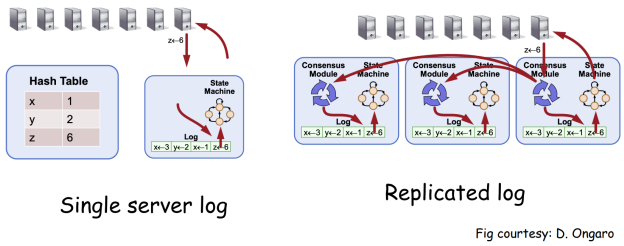


Figure 19.2: Example of Log replication

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. Raft is a leader-based consensus protocol

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.

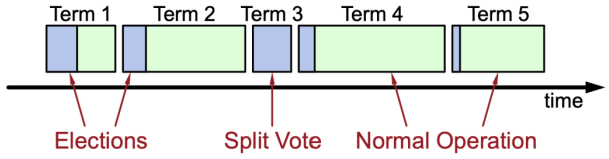


Figure 19.3: Terms

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

Q: What is a split: in network partition, multiple followers become candidates and no one of them can get elected, we need to restart the 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.

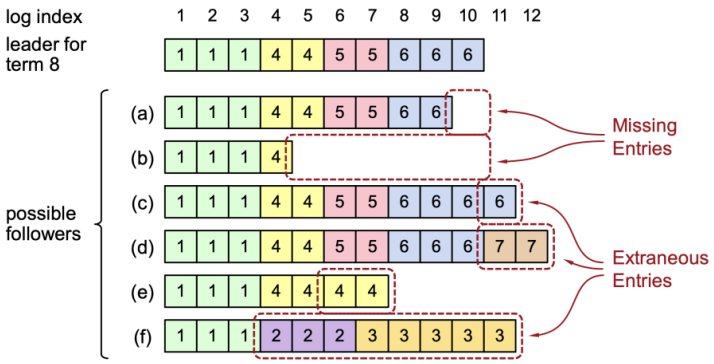


Figure 19.4: Inconsistencies in Logs Example

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.

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.

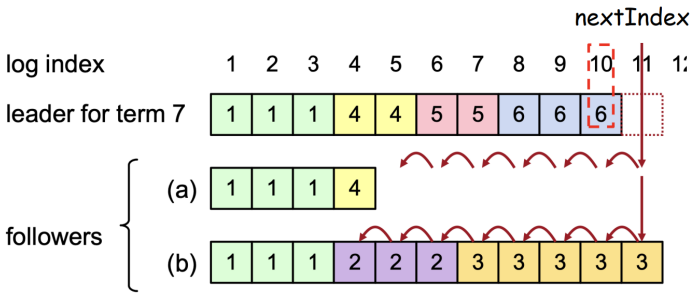


Figure 19.5: Log Repair Example

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.

Q: Is there there any situation when we have match entry, but un-match entry before: Because the prefix always matches, and we always repair the log to make sure we have the prefix always matches, so there is no case when there will be entries that are not match before the latest matching entry

Q: Why using RPC: No specific reason. Can use different protocol

Question: Why do we have these extraneous entries at all ?

Answer: Assume there was a green leader at some point in time it produced three entries and then maybe there was a network partition , so that green leader and some nodes got disconnected from the majority and before we could realize that maybe that green leader sent an extra entry to this follower but it can't send it to the remaining majority but they are already in an unreachable so they will then what they will do is they will elect their own new leader which in this case was yellow and blue and so on so in the network rejoins you will see that some bad things may happen to the minority because before they realize that something had gone wrong they had already written some entries to their law okay and you've got to repair them.

Question: What if the step that is extraneous was a really important operation and we deleted them how is whoever made that request is going to figure this out?

Answer: In this case, because the leader got disconnected, it couldn't gather the majority vote and hence it never replied to client with a success response. So the client so the request will simply timeout or fail.

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.

Lecture 20:Consensus

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.

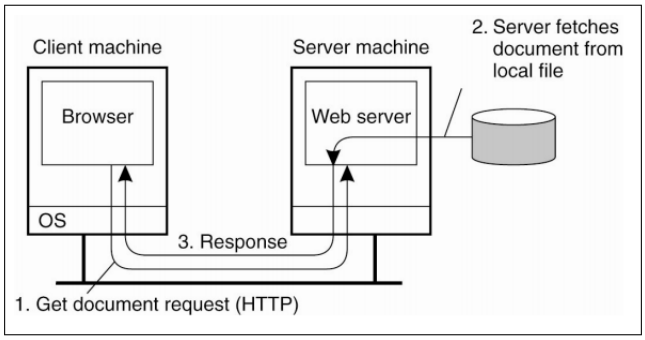


Figure 20.1: Overall Organization of a Traditional Web Site

20.2 Web Browser Client

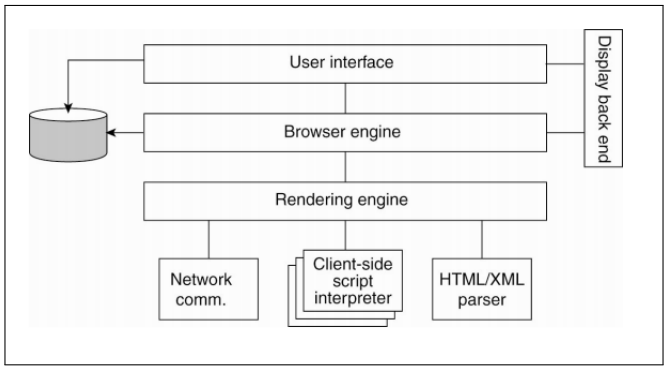


Figure 20.2: Logical Components of a Web browser

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 such as a

network component for network communication, client side scripts for interpretation (example: Javascript interpretation), and a parser that can parse HTML/XML.

20.3 Apache Web Server

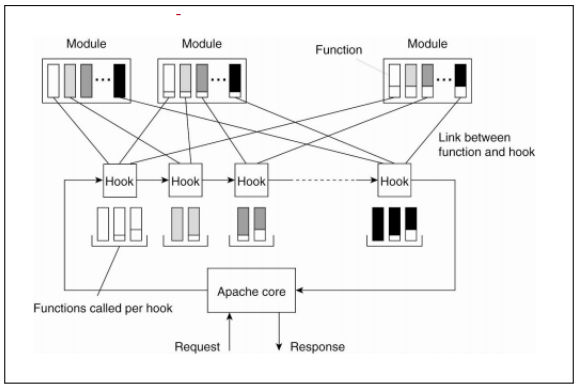


Figure 20.3: Overall Organization of Apache Web Server

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. 'Hooks' are used to call these modules. 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 and content translation (example: While trying to watch a video on a phone, a server might have the video with a resolution higher than is supported by the phone screen. A proxy can translate the high-resolution video to a lower resolution supported by the phone). 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.

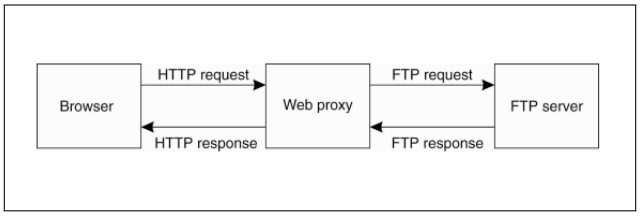


Figure 20.4: Web Proxy

20.5 Mutlitiered Architecture

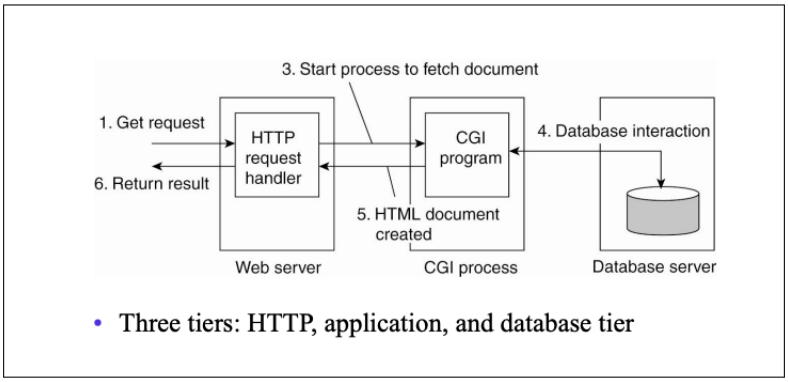


Figure 20.5: Multitiered architecture

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.

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:

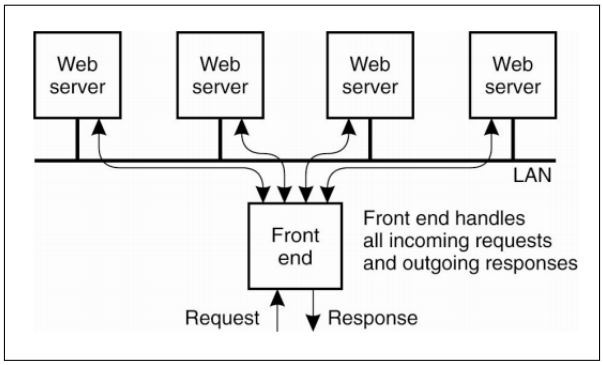


Figure 20.6: Web Server Clusters

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. Only the web tier and app tier are replicated. 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) sends the request to a distributor. The distributor's goal is to do load balancing. The distributor may also communicate with the dispatcher whether to handle the request on the current server or if it 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 response comes back, the load balancer appends this to the first request and sends it back. These are essentially 2 TCP 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.

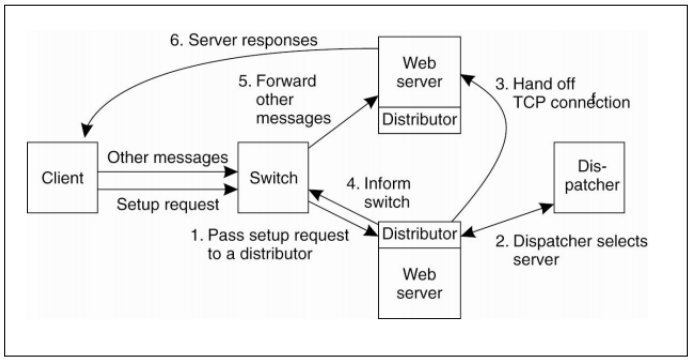


Figure 20.7: Scalable Content-Aware Cluster of Web servers

Question: In TCP splicing, will the front-end keep the connection alive or construct/deconstruct?
Answer: HTTP redirect, TCP Splicing and TCP handoff are independent of whether connections are persistent or not. In either of the two cases, all these techniques can be applied.

Question: Does TCP splicing have an advantage over TCP handoff?
Answer: In TCP handoff, the request is completely handed off to another server whereas in TCP splicing there is a middle entity that works as a relay. This relay can become a bottleneck. In TCP splicing the replicas are hidden.

Question: Can the dispatcher be replicated to remove bottleneck?
Answer: To some degree we could. However, the dispatcher might have a state in it. The state also has to be sent to replica for consistency reasons.

Question: What is the actual path of request in an HTTP redirect?
Answer: In HTTP redirect, the client first makes an HTTP connection to the front end. Once the front end does an HTTP redirect, the request goes to one of the servers. At that point, the client is directly talking to the server.

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: Why is the dispatcher sitting separately from the switch?
Answer: This is an older architecture where the switch directly forwards the request to the web server and the server's distributor talks to dispatcher to determine whether to handle request on the same machine or forward the request to another machine. In more recent architectures, the dispatcher sits along with the switch to directly forward the request to the right web server.

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 or decreasing 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 number of the clusters, but change the size of the replicas, by giving or removing cores.

This is used widely in modern cloud-based applications.

When do we scale?

1. Proactive Scaling: Look ahead and predict workloads (maybe for the next hour) and pre-provision resources ahead of time.
2. 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 the time taken to start the machines.

Question: If there's a web application that is using a private cloud and its workload goes up, will it overflow in the public cloud?
Answer: It depends on the current utilization of the available resources (private/public/hybrid cloud). If private cloud has available machines there is no need to go to a public cloud. If there are no machines in the private cloud, then the application can be overflown into public loud. In such a case the load balancer

has to be intelligent enough to determine which requests to send to the private cloud and which to send to the public cloud. Elastic scaling techniques apply in all of the above scenarios.

Question: Does elastic scaling deal with workload increase due to malicious requests or web crawlers?
Answer: Elastic Scaling doesn't look at the cause of the workload increase. It will simply increase the capacity in the case of increased workload.

Question: In vertical scaling, can we increase the number of resources without restarting?
Answer: It can be done with specific implementation of VMs. for eg. a VM can have 10 virtual cores mapped to 2 physical cores and the number of cores can be later increased to 3 without having to restart the VM.

Question: In practice which one is used? Horizontal or vertical scaling?
Answer: Most large services use a combination of the above techniques to ensure that they never run out of capacity. But mostly horizontal scaling is used because it's easier to just start an instance rather than having a special VM for increasing/decreasing the resources allocated.

20.8 Microservices Architecture

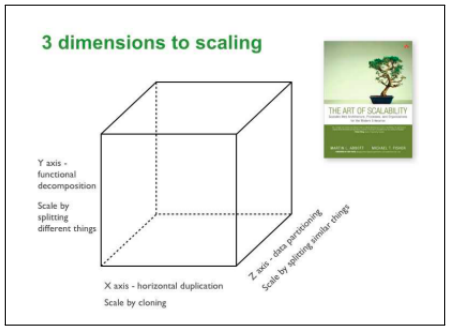


Figure 20.8: Scalability cube which shows 3 ways of scaling

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.

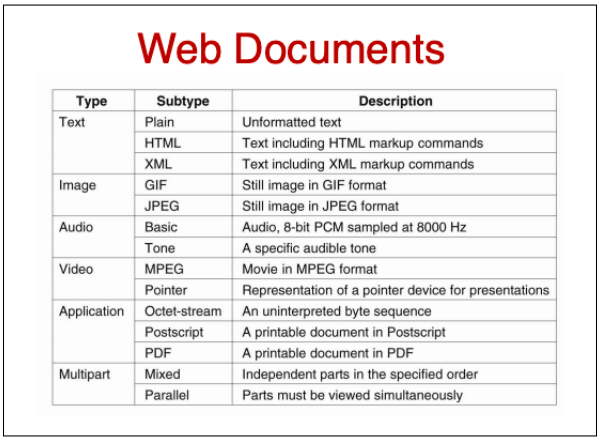
Question: Why is data partitioning a way of scaling?

Answer: Let's say we have an application that has 100,000 users. You can write it in a way where each replica is capable of serving all 100,000 users, or you can write it in a way where each replica handles a fraction of users. That's data partitioning. Each replica is responsible for some subset of users and their data. A request is sent to the right replica that holds the user's data. As users grow, we can repartition these responsibilities.

Question: In data partitioning, are data and servers both partitioned?
Answer: In data partitioning there are multiple servers. Rather than saying that any server can serve any user or access any data, the responsibility is partitioned to specific servers. One server is responsible for a subset of data, another server takes care of a different subset of data, and so on.

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.



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 a 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 the next HTTP requests over the same connection. The browser does not close the connection immediately, in anticipation of further requests. However, these are sequential. We need to get the 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.

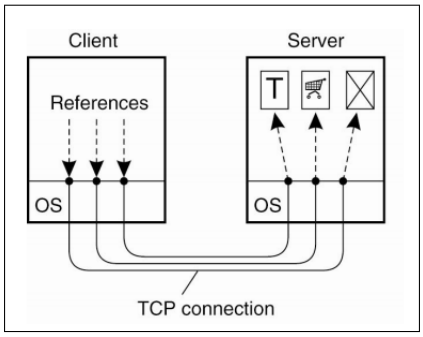


Figure 20.10: HTTP 1.0: Using Non-Persistent Connections.

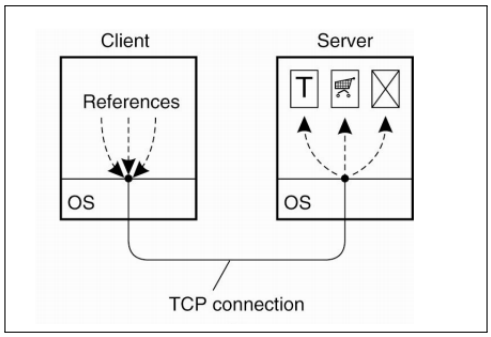


Figure 20.11: HTTP 1.1: Using Persistent Connections.

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.. It is the most used HTTP command.
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.

Question: Is 'PUT' a holdover from FTP or some prior protocol?
Answer: 'PUT' was designed from FTP's perspective, but it is still used for other purposes, although not as much as 'GET' and 'POST'.

Question: What is the difference between 'PUT' and 'POST'?
Answer: 'PUT' is used to send entire files to the server. 'POST' is generally used to send a part of the data but not the entire data. Web forms are a good example to think of when talking about the 'POST' HTTP method. The fields of the form are sent as individual fields (name, email, etc.) and not as a single entity.

20.11 HTTP 2.0

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. In HTTP 2.0 responses are not guaranteed to be in sequential order. 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.

HTTP 2.0 has bidirectional functionality where the server can push data to the client. This is because HTTP 2.0 is a push based protocol(as compared to HTTP 1.0 which is a pull based protocol). for eg. If the server has active connections to the client, the server can push updates rather than the client polling the server.

Question: Is there a new version that is going to use UDP?
Answer: Yes, there is. HTTP 1.0 and 1.1 only used TCP. HTTP 2.0 can run on different protocols, including UDP. Google version of UDP is QUIC which is used when the chrome browsers connect to google servers. But here all features of TCP such as 3 way handshake are not available and have to be taken care of at the application level.

Question: How is the server designed? Will there be multiple threads for one connection or each connection will have a single thread ?
Answer: This is an implementation detail. HTTP 2.0 does not define this. An application developer has to take care of this.

Question: In HTTP 2.0, are multiple connections preferred over a single connection?
Answer: In HTTP 2.0 the idea is to use one connection to send multiple requests in parallel and get responses in some order. We try not to have multiple connections. We can still use multiple connections, but it is designed to use a single connection.

20.12 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.

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.

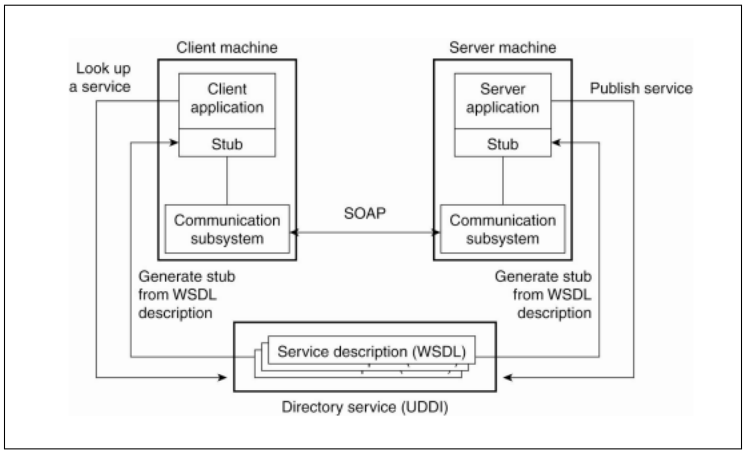


Figure 20.12: Web service

```
<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

20.13 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. Rather than using XML to send data, it uses HTTP to send request and get a response. 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.

Figure 20.14: Example of a Restful web service

20.14 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 web services have a client and a server and are 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 do not have any pre-defined standards. They have general guidelines, but no one particular way to write them.
4. SOAP is very heavyweight compared to REST.
5. Rest has less of a learning curve compared to SOAP.

20.15 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.

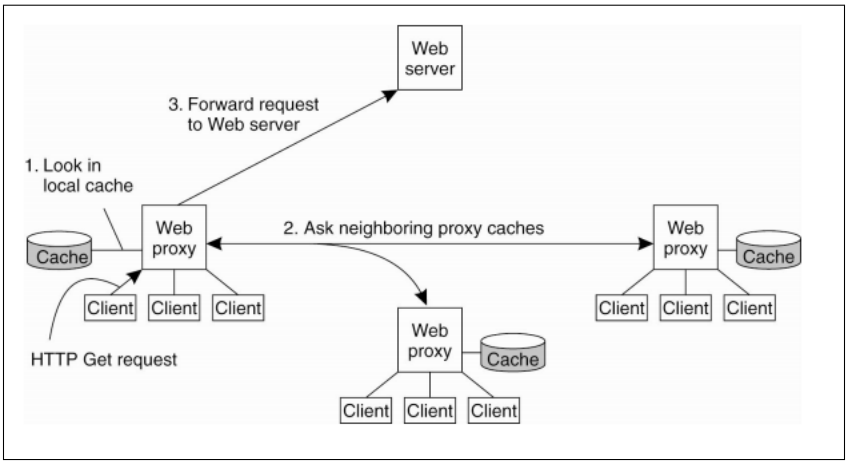


Figure 20.15: Cooperative Caching

20.16 Web Caching

It is also important 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.

Lecture 21:Traditional Web Based Systems

21.1 Web Caching

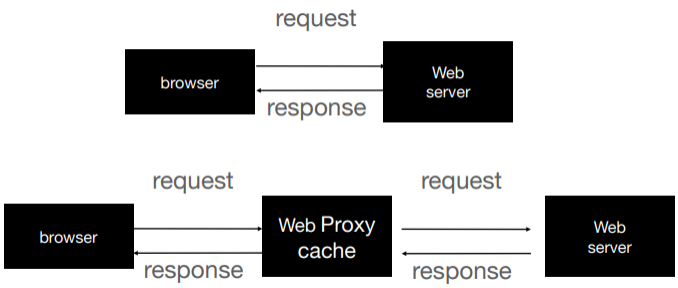


Figure 21.1: Client-Server and Client-Proxy-Server Architectures

Web caching uses a 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, it returns the content. This helps in faster responses for the client, if the proxy is near the client, and reduced loads for the server.

Note: Web pages can either be dynamic or static. Dynamic web pages are regenerated for every request and for every user, as opposed to static web pages which can just be pregenerated HTML files which only change when the user changes them. Therefore, dynamic web pages are typically not cached. For this reason, content being referred to in these notes is largely static content (static audio, video, web pages etc).

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 its 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 a nearby/local proxy instead of the server i.e. it will reach out to the near by proxies to get the data. In this case, all the caches act like one big logical cache, so the clients will see union of all the content stored in nearby caches rather than just the content cached in the local proxy. This can make fetching faster than getting data from the server.

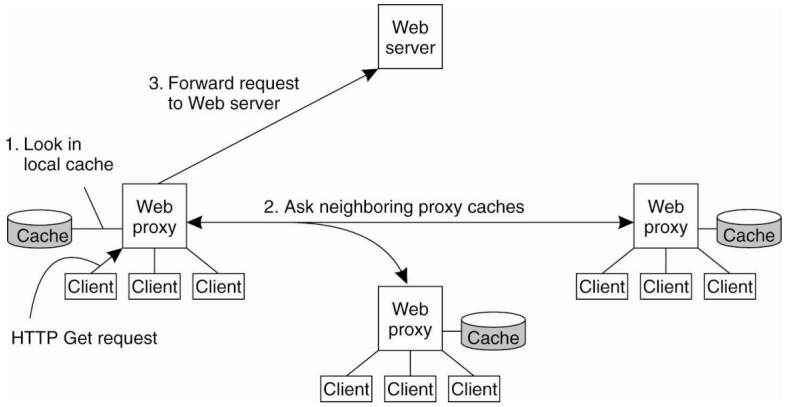


Figure 21.2: Web Proxy Caching

21.3 Consistency Issues

Recall a system has consistency if all the replicas see the same data at the same time. In the context of web caching, this means that when a browser fetches a page, we are guaranteed that the returned page is the most recent version. Web pages tend to change with time, so if we cached a webpage and the webpage changed on the server, the proxy may serve stale content. So, we need to ensure the consistency of cache web pages.

In tackling the issue of consistency, we need to in consideration, the the read frequency (how popular the web page is) and the update frequency (how often it changes). There are 2 approaches for maintaining consistency - pull-based and push-based.

21.3.1 Push based Approach

This approach relies on the server, i.e. it is the responsibility of the server to ensure that the content stored at the proxy caches is up to date. For every web page stored at the server, the server keeps a table of all proxies that have a cached copy of that web page. Whenever a page is updated, the server looks at the table, finds all the proxies that have a copy of the web page and notifies each of them that the page has been updated. This notification can be of 2 types -

1. **Invalidate** - inform the proxy that the web page has changed (so it can discard the page from its cache).
2. **Update** - send the new version of the page. This tells the proxy that the page was changed and that it should replace the old version (in its cache) with the new one.

When to use invalidate and when to use the update message?
This depends on the read and update frequency of the content. Invalidate is a relatively small message whereas an Update can be large (since it contains the content of the entire web page). If the content is unpopular (has a low read frequency) then it makes more sense to send an invalidate message, since many proxies may not need the new copy (and so sending it wastes bandwidth). On the other hand, if the web page is popular then most proxies will request the updated web page after getting an invalidate message. In this case, simply sending the copy of the web page in the first place would have saved bandwidth.

Advantages of a push-based approach are that it provides tight consistency guarantees and that proxies can be passive, as the server does all the work in this approach. Disadvantages are that the server becomes stateful - for every web page it keeps track of proxies. So, since HTTP is stateless, you need mechanisms beyond HTTP. Moreover, the server has to keep track of proxies for a long period of time (possibly indefinitely) since it can't know when to remove them.

Question: While the invalidate message is in transit, the cache is already stale. How to deal with this?
Answer: We cannot deal with this issue because even if the server instantly sends the message there is still a speed of light propagation delay for the message to reach the proxy. For this period of time, the proxy is going to have stale code. Strict consistency in practice is hard to achieve.

21.3.2 Pull based Approach

This relies on the proxy 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 web page only if it has changed.

When to poll for web pages?
Depends on the frequency of updates. 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). 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 web page 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 the average rate at which web page is changing.

Question: Why not just poll when you get the request?
Answer: We could do this, this is the only way we get strong consistency because when a request comes and it's a cache hit. But we don't know if it's consistent, so we can do an If-modified-since request to check if the cache still consistent.

Question: What is the downside of the approach mentioned in the above question?
Answer: Increase in latency. The reason we are caching the content at the proxy is that proxies are close to the client and they can deliver content faster. However, before delivering that content every time if you have to go to the server to check, it introduces another round trip time delay. You may as well go to the server and fetch the content directly.

Question: In the pull-based approach, does the server have additional overhead because of the additional If-modified-since http messages?
Answer: Yes, there is additional overhead because these checks may need to be made frequently to maintain consistency.

Question: Does DNS use a pull-based approach?
Answer: DNS is a pull-based protocol in the sense that it translates hostnames to IP addresses, and the browser sends it a hostname for which the DNS returns the IP address. DNS also uses caching, for which it just uses TTL values (since IP addresses don't change that frequently).

Question: Why can't we do a hybrid of both approaches (Pull-based and Push-based)? Why can't the developer then decide, or the proxy decide dynamically?
Answer: The server has to support a push-based approach. For example, if the server was just a standard http server then it won't have any push-based functionality. If the server does, then you could do a hybrid approach.

Question: Do we need to worry about clock synchronization?
Answer: Not too much, assume clocks use NTP accuracy of 10 milliseconds. As web pages won't change every 10 milliseconds (usually changed in days or weeks), we need not worry about synchronization.

Question: Polling is to keep the content in the cache consistent with the server, but how does the proxy know whether to keep the cached content at all? Maybe nobody is interested in it.
Answer: To understand when a proxy should maintain cache and when not to it has to track some statistics (like the rate a page is requested at). If the request rate is very low, the proxy can decide the page is no longer popular and evict it. On the other hand, if the request rate for a page is high proxy will decide to keep it consistent.

Question: Pull-based approach does not have persistent HTTP connections, is that a limitation of this approach? And does server push have 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.

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 synchronize the content might overtake the benefits of proxies. There is a higher overhead than the 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.

21.3.3 A Hybrid Approach: Leases

Hybrid approach based on both push and pull. Figure 21.3 illustrates how it works.

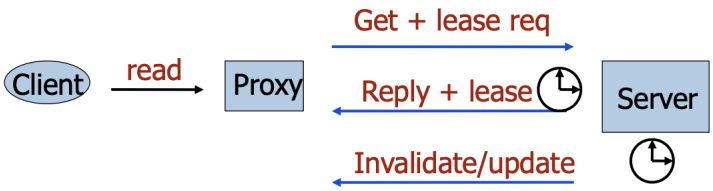


Figure 21.3: Lease

Lease is a contract between two entities (here the server and the proxy). It specifies the duration of time the server agrees to notify the proxy about any updates on the web page. Updates are no 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, it degenerates to polling (pull). If duration is infinite, it is the same as push-based 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 to 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: What is age-based lease?
Answer: First, what does age mean here? The age of a file or web content is the time since it was modified last i.e. time now - time file was last modified. If a file has old age it means it hasn't been modified and a younger file means they've been modified recently. We can use age to decide how long a lease should be.

Question: If the file is modified more frequently, the age will be lower so the lease time will be longer?
Answer: In the slide it says to give longer leases to the objects with larger lifetimes, but you can do exactly the opposite of this depends if you want to reduce the workload on the server or not. Giving long leases for frequently updated objects increases the server load.

Question: If a proxy evicts an object from the lease, can the lease be canceled? **Answer:** Original lease mechanism does not have any cancellation mechanism, but you can add one. Ideally, if you have an active lease you should not evict an object but if you do for any reason, there has to be either a way to cancel the lease or you'll get some wasted notifications from the server.

Question: Does each proxy have its own lease?
Answer: Not only does each proxy have a lease, there is a lease for each web page at a proxy. If a proxy caches 100 different web pages, each of them will have a different lease. It is not a lease per proxy, it is a lease per web page 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 web page changes for every active lease on that web page. 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. Server can know the age and server load. Proxies can track popularity of the web page and report stats to the server. Monitoring framework can be added to the proxy system, it is not a part of the lease approach.

21.3.5 Cooperative Caching

Recall, in cooperative caching a collection of proxies cooperate with each other to service client requests. These proxies can be arranged in different structures. In "Hierarchical Proxy Caching", the proxies are arranged in a hierarchy, a tree-like structure, where the server is the root and the rest of the nodes are caches.

Figure 21.4 shows an example of this. The client sends a request to one of the proxies. If it is a cache miss, the proxy will send requests to its 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 forward the HTTP request to its parent and the whole process will recurse until a cache hit occurs or until the server is reached. The data will then be sent back as response - and will flow down the hierarchy, back to the client.

Question: Why are we using ICP instead of HTTP?
Answer: ICP is designed specifically for caching and cache consistency, whereas HTTP is not. ICP is basically just a way to ask other nodes if they have some content and fetch the content if they do, so its not very different to HTTP in that sense.

Question: What if the proxies have different versions of the file?
Answer: Here, the assumption is that if any one proxy has the content, then that content can be sent. To maintain consistency, whenever a proxy knows that the content has changed it can inform other proxies of the latest version.

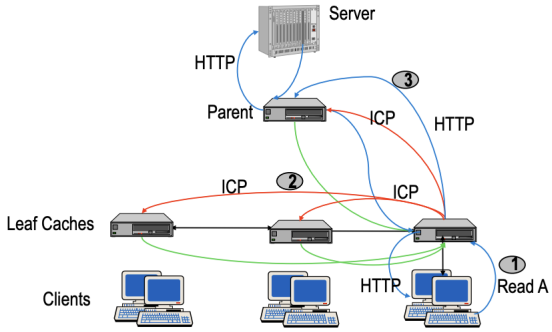


Figure 21.4: Hierarchical Proxy Caching

This works well when a nearby proxy actually has the content. If there is a global miss - no one in the hierarchy has the content, latency will increase significantly. Clearly, 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 the server in the event of a cache miss on the proxy.

To address this problem, we'll look at a different approach for doing this. Every time we send a request up the chain, it adds more hops which increases the overhead and thus the latency, so firstly we'll remove the hierarchy. This gives us a flat structure where every proxy directly communicates with all other proxies. If the content for a request is in a nearby cache, it is fetched. Otherwise, the content is fetched directly from the server.

Moreover, we want to reduce the overhead of querying other proxies to see if they have a page or not. To achieve this, every proxy keeps track of what is stored in its nearby caches. Now, whenever a request comes in, if the proxy doesn't have the data in its cache, it looks in the table to see if any nearby cache has it. And likewise, if there is no entry in the table for a request i.e. none of the nearby caches have the content, it gets it from the server and adds it to the table.

Using this approach, lookup is local. A hit is at most 2 hops and a miss is also at most 2 hops (as opposed to 1 hop in the other method). Every time the proxy fetches or deletes a page, it updates the table for all

the nodes. Every proxy keeps a global table that must be kept consistent. There is an additional overhead for keeping this consistent, but the performance is better.

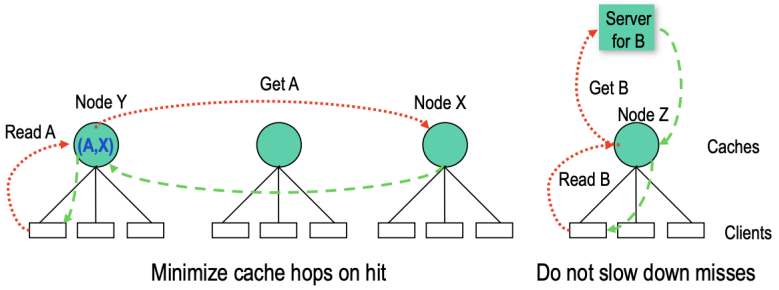


Figure 21.5: Locating and Accessing Data in the Flattened Network

Question: Do we force the miss by enforcing that it goes to at most two hops?
Answer: Yes, two is the most we will have because there is no re-forwarding. If a request comes in from another proxy, we will not check if some other proxy has it. So, a request from a client and one from another proxy are treated differently.

Question: Is this approach going to be more efficient than the previous one? Because we have to inform all other caches after every cache update.
Answer: There are two things to keep in mind. Firstly, the caches are not going to change that frequently. Secondly, the purpose of having proxies is that the latency times for clients are reduced. Updates can happen in the background, so they don't add to clients' latency.

Question: Why could we not use Hierarchical Proxy Caching if there are only popular web pages?
Answer: We cannot assume users will only request for popular pages they can ask for any page.

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 independent of whether a lease is being used or not.

Follow-up Question: If you do multicast, don't you require a 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.

Question: If you want to do multicast, how do you identify proxies that have the content?
Answer: You would have to construct a multicast group for every web page. When a proxy caches that content, you put that proxy in that group. When the content is removed from the proxy, you remove the proxy from the group.

Question: If you have a hierarchical 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 distant 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. Computationally intensive tasks were 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 many these servers in many different networks. Servers are deployed as clusters of different sizes depending on the demand.

Content providers are customers of the CDN service. They decide what content to cache and it is their responsibility to maintain consistency.

Users access website normally, the content is fetched by the browser from CDN cache.

CDN Request Processing

Figure 21.6 shows how a request is routed from a client to a CDN server. The client is going to go the server and make a request for an html page (this is not cached, so it comes from the origin server). Within the page, there may be content; the URL for that content will point to one of the CDN servers.

Now, the CDN has to decide which of its caches will give the content. This is done through a smart DNS lookup. Recall DNS ("Domain Name Service") is a service which takes the url and gives the IP address. The browser makes a connection to this returned IP address.

In a smart DNS lookup, the DNS Server will check where the client is located and return the IP address of the nearest cache.

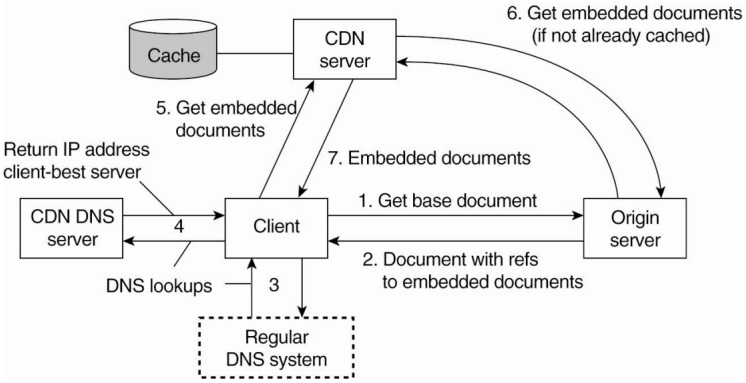


Figure 21.6: CDN Request Processing

CDN Request Routing

When a CDN gets a request, it needs to send to the nearest cache (to get the lowest latency), so how does it decide which proxy has to serve that request? They have large load balancers that look at incoming requests and send it to different proxies. Typical CDNs have 2 level load balancer:

1. Global level - Which cluster to send the request to. This is done using DNS (as described above).
2. Local level - Once the request is mapped to a cluster, which server in the cluster will serve the request? (When a request is routed to a cache, it will usually not be a single cache, but a cluster of caches).

Question: If it's a local load balancer does it use concepts like least loaded, round robin?
Answer: Yes, that is exactly what the local load balancer does. As long as content is replicated, you send it to any of the caches; if its not you have to look at the url and only send it to the subset of them that have it.

Question: Do edge proxies use cooperative caching?
Answer: In this case, no need to do cooperative caching. Essentially replicating 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, 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.

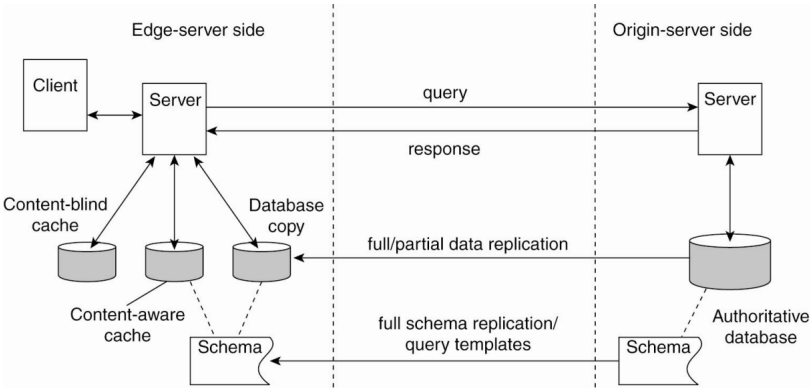


Figure 21.7: CDN Hosting web apps

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.

Lecture 22:Web Caching

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).

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.

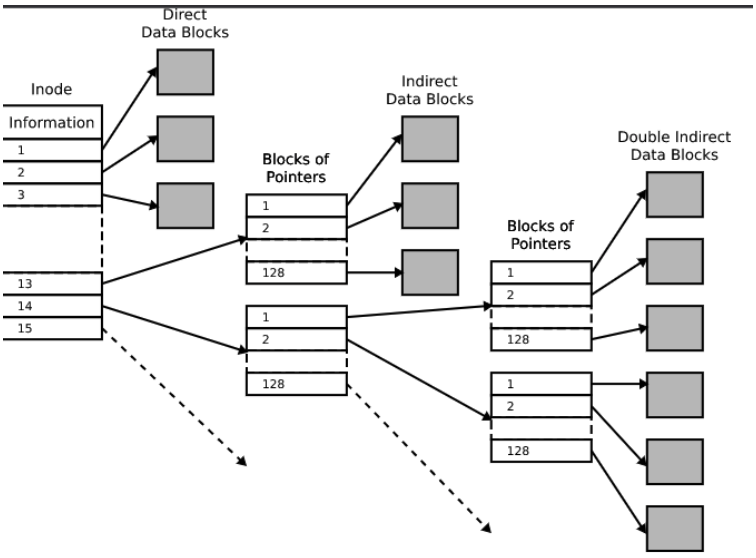


Figure 22.1: Inode structure

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. This model is typically stateful since we need to keep track of which clients are accessing which file and so on. This might eventually cause the server to become a bottleneck if there are many incoming requests from multiple clients (I/O bottleneck at the server).
- 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. This model works only if there is one client one file at a time, hence maintaining consistency.

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.

Question: Which model does Google Docs use?
Answer: Google docs is a cloud service and not a distributed file system in a technical sense. Today, Google, One drive, Dropbox provide a form of remote storage that looks like distributed file system, but is not necessarily the same. Answering the question, google docs model depends on the mode of the browser.

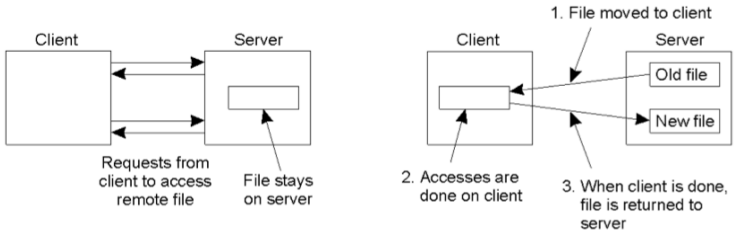


Figure 22.2: Remote access model(left), Upload/download model(right).

In the online version, every change is saved on the cloud server instantly, while in the offline mode, there is a copy at the client and a master copy at the server.

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). Here, the client and server communicate with each otehr using RPC calls. The VFS layer in the client and server provides a system independent abstraction to the layer above it. Thus, no need to worry about the type of the filesystem the file is stored on.

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 needed whereas in version 4 of NFS, we can perform a batch RPC request. Version 3 executes one RPC per operation, whereas version 4 supports multiple RPC calls per operation(batched).

Question: Is the NFS Client and Server implemented as a user space process or is it implemented inside the kernel?

Answer: It is an in kernel implementation. It is not a user space process. In most operating systems, NFS code is going to reside inside the kernel like any other file system code but this is just distributed in nature.

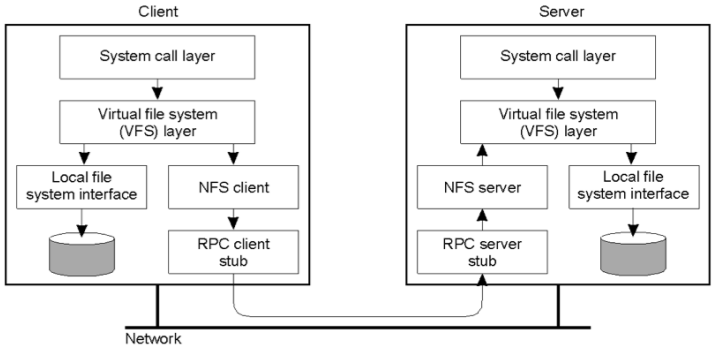


Figure 22.3: Network File System (NFS)

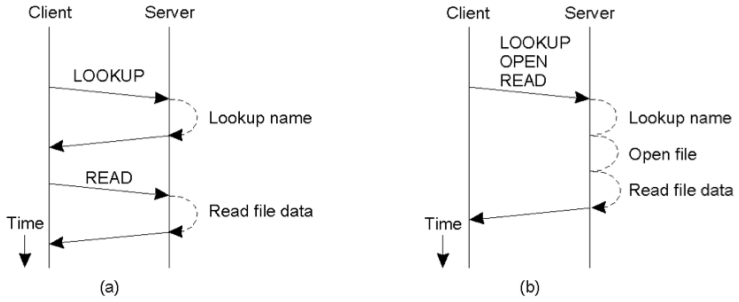


Figure 22.4: Difference in communication in NFS version 3 and version4.

Question: Why is Lookup triggered?

Answer: We usually access a file by opening the file and reading it. This eventually will go to the OS as a system call that's called an open system call at the OS level. This will go to a virtual file system layer and then the NFS Client and client has to service the open but version3 does not have a concept of an open so instead it is going to send a lookup operation to the server saying client wants to access this file. Then checks if file name is valid and if client has privileges to access it. It sends a response in a yes or no (yes if open call succeeds) then we get another read system call and another RPC is triggered.

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.

Question: Client A can access few files in the mounted directory and Client B can access few files in the mounted directory, can you see all of those files?

Answer: The visibility of these files is controlled by file permissions. Similar to Unix we set the file permissions which mention read write and execution permissions to other users. OS will control this and file sharing can be done in distributed systems in the same way.

Question: Does mounting create a list of files stored on the server or are you always going to lookup files on the server?

Answer: Mounting is simply a mapping. It is not going to create any list. Mounting creates a mapping from the directory searched to the directory's location on the server. Client OS is not going to know what is stored in the directory. So when the user tries to access anything inside the directory, all the operations are sent to the server and whatever responses are returned are checked.

Question: Is concurrent access possible in this model?

Answer: There are two types of concurrent accesses: accessing same file, accessing same volume of files. It is of course possible, example Ed Lab with multiple users accessing the file volume on the system and working on it concurrently. To access the same file, it is possible but we need locking mechanisms to avoid overwriting each other's data.

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.

22.2.11 Client Caching: Delegation

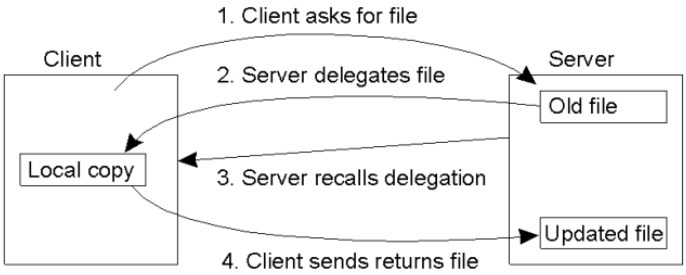


Figure 22.5: Delegation

NFS supports the concept of delegation as part of caching. The client receives a master copy of the file to which the client can make updates. Upon completion, the client can send the file back to the server. This is similar to the concept of upload/download model. Thus, the server is delegating the file to the client so that the client can have a local copy. If another client tries to access the file, the server recalls the delegation given to previous client. The previous client returns the file to the server and then the server uses the old model where multiple clients can access the file by read/write requests to the sever.

Question: When does the server decide to delegate the file?
Answer: Since this feature is stateful, it is only present in version 4. If the server is serving only one client then the server can delegate the file. Otherwise since the server is not the current owner of the file, the server can not delegate and thus has to use the old model. For example, files in the user's home directory can be delegated, whereas binaries of application programs can not be delegated as multiple users might access them.

Question: Is there a way to periodically update the server as in case of client failure the files may get lost?
Answer: It is possible for the client to flush the changes to the server in the background while it still holds the master copy.

22.2.12 RPC Failures

For RPCs being used over TCP, TCP will take care of retransmissions between client and server. For RPCs over UDP, client and server can decide how to deal with lost requests and replies. Every RPC request is associated with an ID. Upon receiving an RPC request from the client, the server will maintain the request and response for that request in its cache. If the client resends the request and the reply was lost, the server will simply send the reply from the cache, instead of executing it again.

Question: What is the utility of UDP over TCP?
Answer: UDP is faster than TCP as there is three-way handshake in TCP. In LANs, where probability of loss is low, RPCs can be sent over UDP. Over WANs or noisy LANs TCP may be preferred.

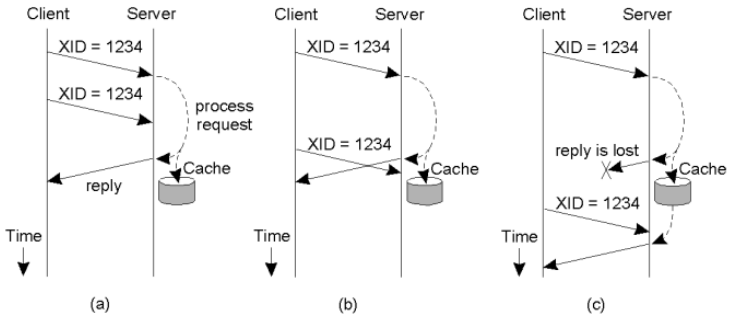


Figure 22.6: RPC Failures

Question: For how long can the reply be kept in the cache?
Answer: It depends on the application. Practically, after some unsuccessful tries within an hour, the client may assume that the server is down. So the replies can be cached for some hours.

Question: Is caching reply specific to some version of NFS?
Answer: It is not specific to some verison of NFS. In NFS v1, there was no concept of RPCs over TCP. Thus, this method was used for RPCs over UDP. Currently, with the advent of RPCs over TCP, this method is not needed to be used.

Question: Is the cache needed only so that the requests are idempotent?
Answer: Yes. For example, if requests are changing files, it might incur problems if they are not idempotent and if requests are needed to be idempotent the cache is required.

Question: Who is responsible for keeping the IDs unique, client or the server?
Answer: The IDs have to be generated at the client. This can be made unique by using the client's IP address followed by a number that is incremented sequentially for each RPC call.

22.2.13 Security

Versions 1, 2 and 3 of NFS relied on a simple security model. Every request is sent with user ID and group ID. The server checks for the file permissions on the basis of user ID and server ID. This ensures only authenticated users can access the file. One drawback of this is that the channel between client and server, however, is not still secure. If an adversary intercepts the network traffic, the contents of a secure file may be exposed. In version 4, the concept of secure RPCs was introduced. Every RPC client stub sends the request to the security layer which encrypts the request before sending. Thus, file contents can not be read on the network.

Question: Client can send user ID and group ID, but how does the server know if it is authentic?
Answer: As long as the server trusts the OS on the client the server knows the user ID and group ID are authentic. However, if the OS is corrupted/hacked the server can not trust the client

22.2.14 Replica Servers

There may be multiple servers serving different set of files. Version 4 allows the files to be replicated. Client can make request for accessing files from any of the replicas. NFS provides implementation of maintaining consistency between the replicated servers.

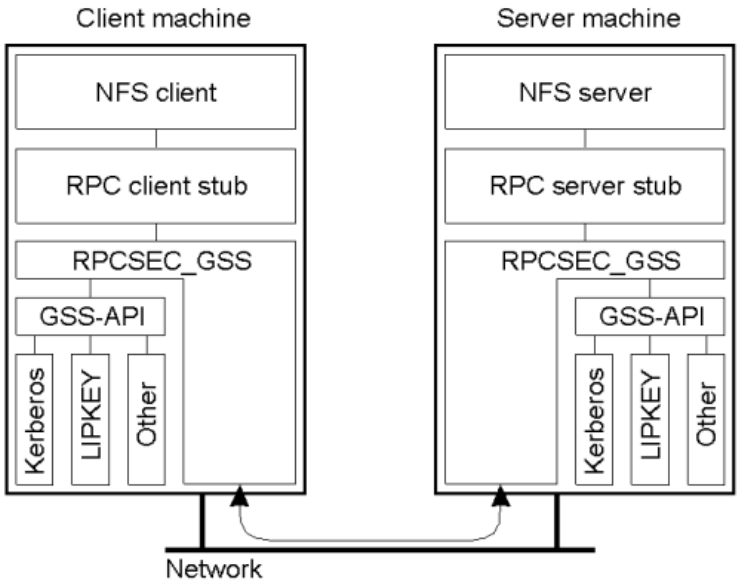


Figure 22.7: Secure RPCs

Lecture 23:File System Basics

23.1 NFS (contd)

23.1.1 Recap

NFS has a weak consistency model. Whenever a client application user modifies a file, the changes get written to the cache at the client machine and later on the client can send the changes to the server. Meanwhile, if the server receives a request for the same file from some other user it will send stale content.

23.1.2 Client Caching: Delegation

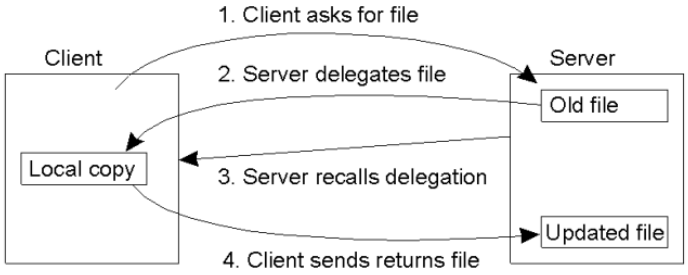


Figure 23.1: Delegation

NFS supports the concept of delegation as part of caching. The client receives a master copy of the file to which the client can make updates. Upon completion, the client can send the file back to the server. This is similar to the concept of upload/download model. Thus, the server is delegating the file to the client so that the client can have a local copy. If another client tries to access the file, the server recalls the delegation given to previous client. The previous client returns the file to the server and then the server uses the old model where multiple clients can access the file by read/write requests to the sever.

Question: When does the server decide to delegate the file?
Answer: Since this feature is stateful, it is only present in version 4. If the server is serving only one client then the server can delegate the file. Otherwise since the server is not the current owner of the file, the server can not delegate and thus has to use the old model. For example, files in the user's home directory can be delegated, whereas binaries of application programs can not be delegated as multiple users might access them.

Question: Is there a way to periodically update the server as in case of client failure the files may get lost?
Answer: It is possible for the client to flush the changes to the server in the background while it still holds the master copy.

23.1.3 RPC Failures

23.2 Coda Overview

23.2.1 DFS designed for mobile clients

- Nice model for mobile clients who are often disconnected

- Introduced in late 80-90s by CMU
- Use file cache to make disconnection transparent
- Designed for weakly connected devices which can be used at home, on the road, away from network connection
- Serves as a precursor to Cloud drives
- It supplements file cache with user preferences. E.g. always keep this file in the cache or Supplement with system learning user behavior.
- Uses replicated writes: Read once and write all. Writes are sent to all accessible replicas.

Question: 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 cache. So when you are disconnected, you're essentially just working with files caches, 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 the mobile device.

23.2.2 File Identifiers

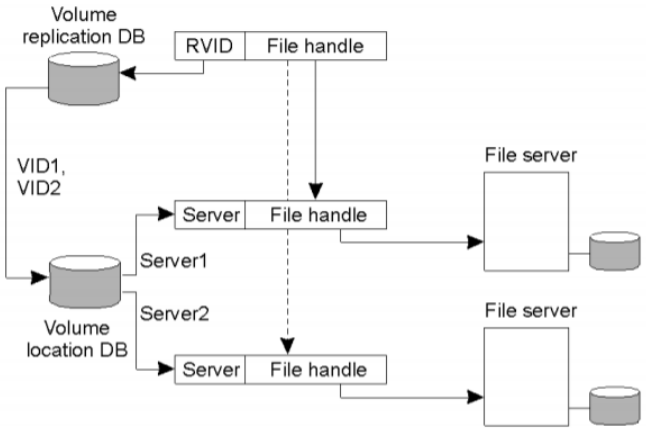


Figure 23.2: Coda architecture

- Each file in Coda belongs to exactly one volume. A volume could be a disk or a partition of a disk.
  - Volume may be replicated across several servers. Identifiers include volume ID and file handle.
  - Multiple logical(replicated) volumes map to the same physical volume

- 96 bit file identifier = 32 bit RVID + 64 bit file handle
- Each write increments the version number. Similar to versions maintained by git.

23.2.3 Server Replication

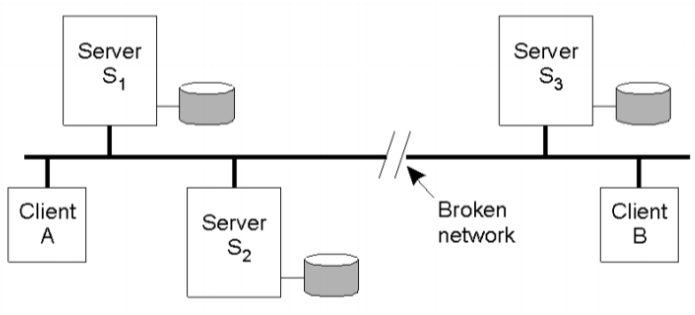


Figure 23.3: Server replication issues in Coda

Assume there are 3 servers and 2 clients connected over a network. In an ideal situation, the servers keep the copies of the files consistent. If there is a partition in the network, the servers no more have the same copy of the files. When network partition is fixed, the servers try to synchronize the files. If the files are different, there may not be any problems. Problem arises if the servers access the same files due to write-write conflicts.

- 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

Question: What is the size of the version vector?
Answer: The number of entries in the version vector is equal to the number of servers that have the copy of the file.

Question: If the file is being updated multiple times will the system keep incrementing the version?
Answer: It is possible. It will still give rise to the same kind of conflict.

Question: What does manual reconciliation mean?
Answer: It means that the user has to manually resolve the conflicts in the same way as the user is required to resolve merge conflicts in Git.

23.2.4 Disconnected Operation : Client disconnects from Server

- **Hoarding State:** Client is connected to the server and is actively downloading files into cache based on some prediction based on current usage of the user. The client tries to cache copies that the user is likely to access.
- **Emulation State:** When the client is disconnected from the server/internet and uses the cached copies.

- **Reintegration State:** Client is connected to the internet/server and merges its updates with the server.

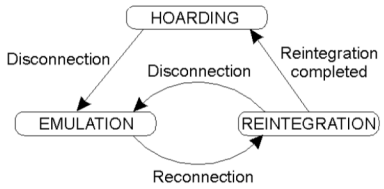


Figure 23.4: Disconnected operation in Coda

- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection.
  - Prefetch all files that may be accessed and cache(hoard) locally
  - if AVSG=0, go to emulation mode and reintegrate upon reconnection

23.2.5 Transactional Semantics

- Network 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

23.2.6 Client Caching

- Cache consistency maintained using callbacks

23.3 xFS

23.3.1 Overview of 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
- All nodes participates in the File sharing

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.

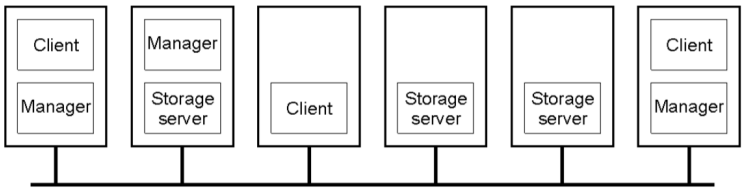


Figure 23.5: An example of nodes in xFS

23.3.2 RAID : 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.

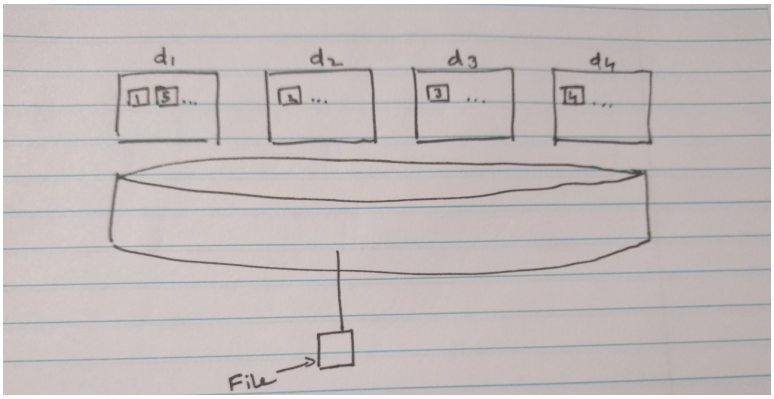


Figure 23.6: Striping in RAID

Figure 23.6 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.

MTTF : Mean time to failure. It is about 5-6 years for a disk. A typical disk lasts for 50,000 hours which is also known as the Disk MTTF. As we add disks to the system, the MTTF drops as disk failures are independent.

$$\text{Reliability of N disks} = \text{Reliability of 1 disk} \div N \tag{23.1}$$

$$\text{Probability of failure of a system} = (\text{1-p})^n \tag{23.2}$$

Consider a case where there are 70 disks in the system.

$$\text{Reliability of system} = 50,\ 000 \text{ Hours} \div 70 disks = 700 \text{ hours}$$

Disk system MTTF: Drops from 6 years to 1 month!

Advantages

- Load balanced across multiple disks
- Parallelizes the access to each disk and hence high throughput.

Disadvantages

- If a single disk fails, 1/N of the data of each file will be lost, without redundancy.
- The performance of this system depends on the reliability of disks.

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:

RAID 0

Doesn't have any redundancy. Only striping. Each files in stripped into multiple parts and stores on a separate disk.

RAID 1 (Mirroring)

From figure 23.7, 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.

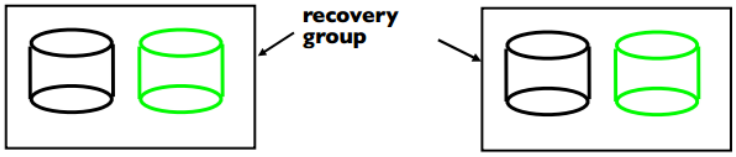


Figure 23.7: RAID 1

RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 23.8. 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:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{23.3}$$

If any disk fails, then the corresponding block can be reconstructed using parity. For example:

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{23.4}$$

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

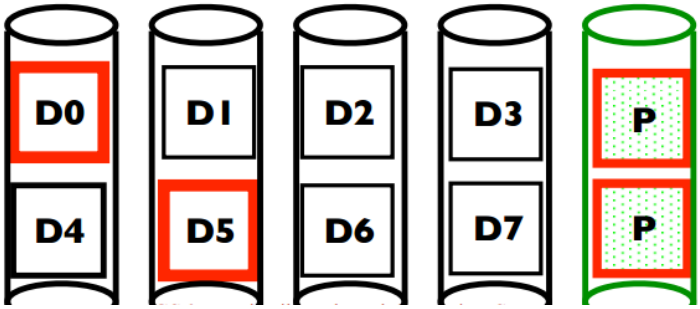


Figure 23.8: RAID 4

group. Smaller writes are expensive, as the corresponding parity would need to be changed and it would require reading the other members of the parity group. Writes seen by the parity disk is N times the writes seen by the other disks.

Question: Where is the information about which files are on which disk?
Answer: The hardware controller serves the request internally to identify which blocks are stored on which disk.

Question: In RAID, hardware controller keeps a track of data blocks and parity, what happens if controller fails?
Answer: There will be problems in accessing the disk. That may be a point of failure. In case of Software RAID this issue will not occur.

Question: Won't the cost of accessing files increase since all disks are being accessed?
Answer: There are two ways to access a file, either block by block or accessing the whole file. If a request is made to access the whole file, in the above figure, eight requests to different disks would have been made, making it parallel. If the entire file would have been saved on the same disk, it would have resulted in eight requests being made to the single disk, which makes it sequential. Thus, accessing multiple disks does not necessarily make it expensive.

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 23.9. 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.

Question: Can a file not be stored in the same disk as its parity?
Answer: The parity and the file stripe wouldn't be in the same parity group. It can still handle 1 disk failure.

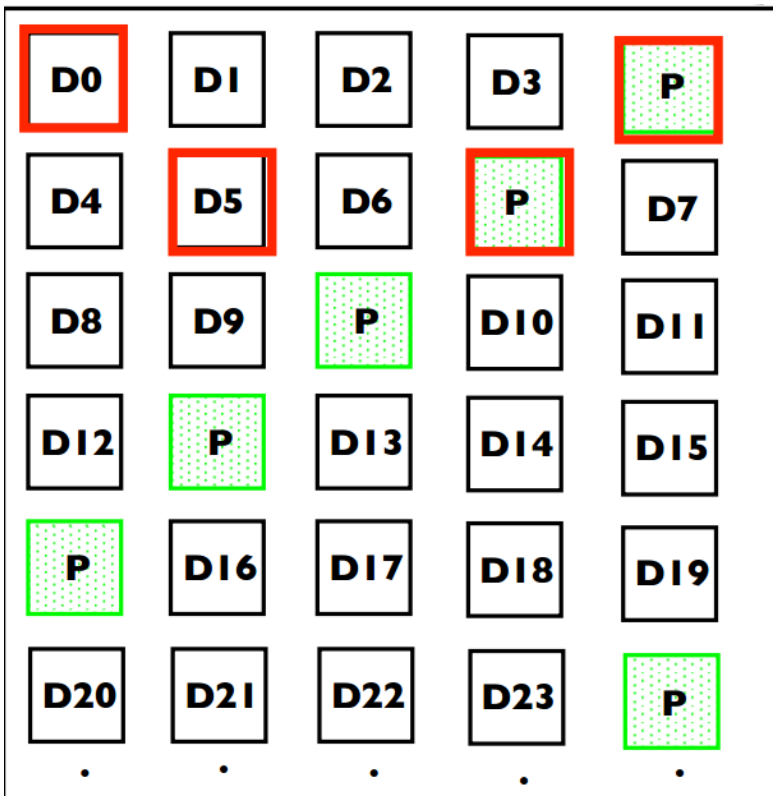


Figure 23.9: RAID 5

RAID Summary

- 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

23.3.3 LFS: Log File Structure

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 i.e. 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.

Question: Is there an overhead to maintain lookup as block of the files need to be tracked?
Answer: There is higher overhead to maintain the lookup. For every write, the data gets appended, so it is meant to be for high write workloads. Metadata of the files is also written to the log. In case of lookups, the metadata has to be accessed. Hence there is high overhead.

Question: Can the writes be cached?
Answer: Reads are directly cached. Writes are cached in batches i.e. a batch of writes are written as an append-only log.

Question: Is LFS one server?

Answer: LFS are traditionally designed as single disk system. Here, they are combined with xFS. The logs are stripped across machines.

Log-structured FS Summary

- Provide fast writes, simple recovery, 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 xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping.
- 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.

23.3.5 xFS uses software RAID and LFS

- 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

23.3.6 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. Optimized for read only applications.
- 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

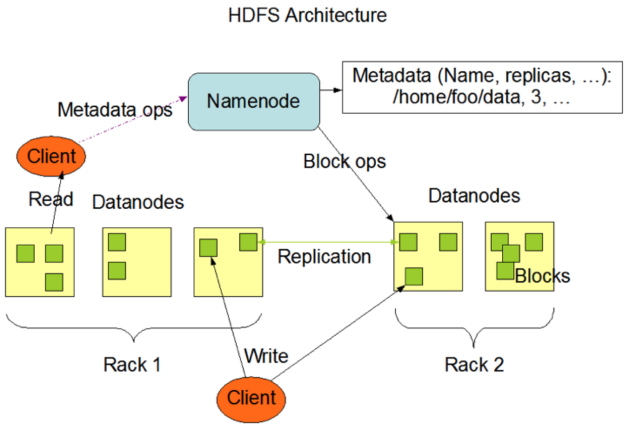


Figure 23.10: HDFS Architecture

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 namespace

Lecture 24:NFS

24.1 NFS (contd)

24.2 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.

24.3 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. The EJB class encodes the business logic of the application. EJB helps in persisting state of objects to the disk and retrieve when needed.

24.3.1 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. Two important attributes: 1. Session 2. Session state is stored on the server side.
- **Entity beans** - They look more like standard Java objects. The object has state which is persisted on disk
- **Message-driven beans** - They are designed for messaging and the messages can persist.

24.4 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.

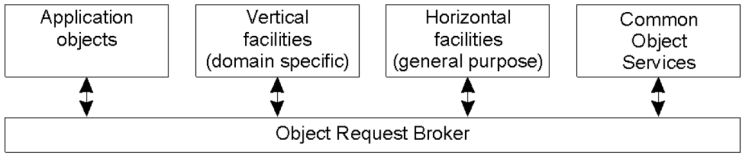


Figure 24.1: CORBA

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. EJB are widely used.

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. It also allows dynamic invocation of interfaces. At runtime, CORBA can fetch interfaces to put in the stub which can then be used. 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.

24.4.1 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. Event channel can be thought of as a buffer. In pull-pull model, event channel polls data from the publisher and similarly, consumer pulls data from the event channel.

Two ways to implement the event channel: 1. Push based model. 2. Pull based model.

We can have hybrid models as well where we can have combination of push and pull models.

24.4.2 Messaging - Async method Invocation

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.

24.4.3 Messaging - Polling based model

In this, the consumer keeps polling the event channel to check if there is any data that aligns with the subscription.

24.5 DCOM : Distributed Component Object Model

DCOM is Microsoft's middleware which has now evolved into .NET. DCOM will only run on Windows servers or desktops. 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. There is a type library which is similar to CORBA's interface library. We also have a SCM (Service Control Manager) which keeps track of what all objects are in the system and where are they running. DCOM is not as heavy as CORBA.

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.

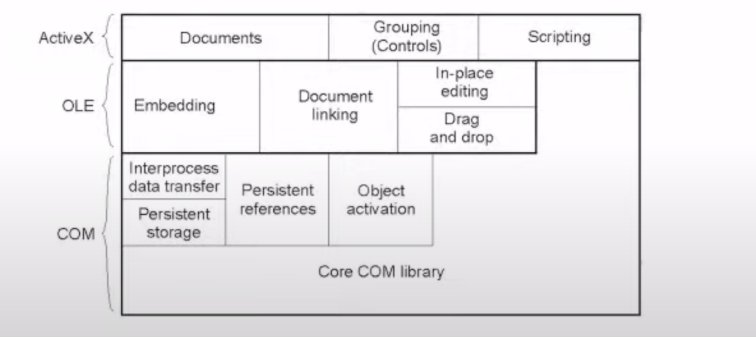


Figure 24.2: DCOM

24.6 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.

Question: If the applications are coupled in time, how does the clock synchronization work?
Answer: Coupled in time does not necessitate that clocks have to be synchronized. It just means that both parties need to be active at the same time. It could be two people or two processes.

24.6.1 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. JavaSpace is basically a database. Each tuple is a Java object. We have reads and writes into a shared database. One can request callbacks in this model as well.

Question : How is this different from a Publish-Subscribe Model?
Answer : In Pub-Sub, subscriptions are done beforehand. In addition, messages posted in the bulletin-board model can be stored for a long period of time.

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. Here the messages are persisted on disk unlike Event channel where messages don't persist.

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.

Question: Why can there be multiple copies of data on the bulletin board?
Answer: That was just shown as a logical view. In reality, if the board is replicated, we write it to all the boards. The logical view just shows a union of all replicas

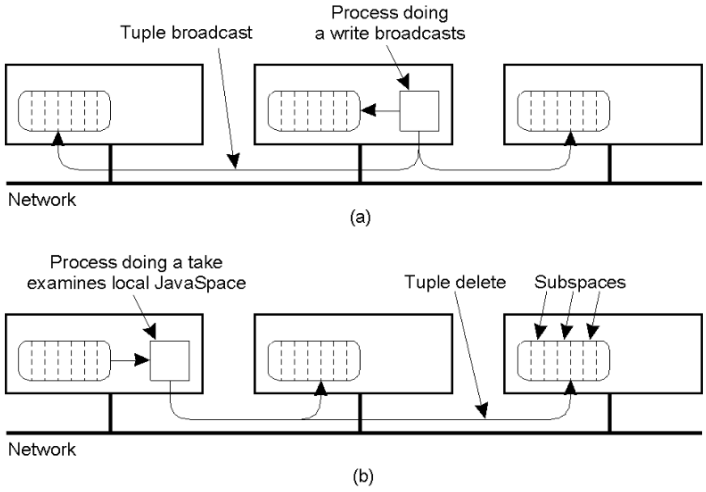


Figure 24.3: Jini Processes

Question: Since messages are not intended for any one process, how does this process(in the example) who read C know that it has to delete C from the database?
Answer: It depends on the number of recipients of a message. If its just one recipient, then the first time a recipient comes and looks for C, we can delete C. If many recipient, then we wont remove it.

24.7 Distributed Middleware Systems

These are middleware systems that are designed for large scale data processing.

24.7.1 Big Data Applications

This is mostly covered in CS532 systems for data science. So we will only partially cover this. We have to parallelized data processing using distributed systems as its a large amount of data.

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.

24.7.2 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.

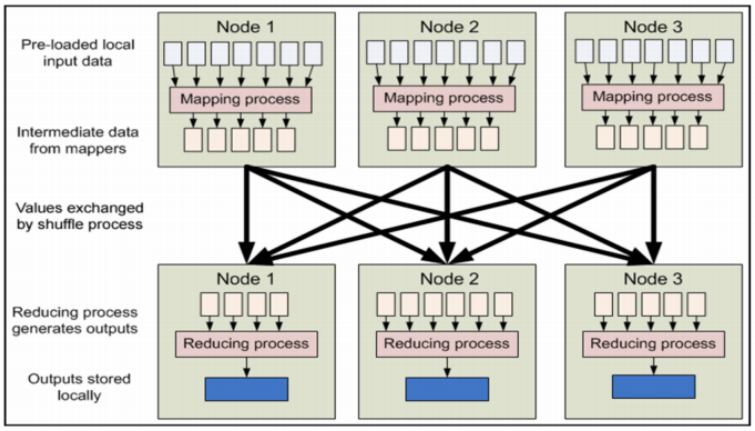


Figure 24.4: Map Reduce example

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.

Example1: Let's say we want to do word counting on a large set of documents. A chunk of data is given to each machine which counts the words within that chunk (map phase). Now we need to sum these up. This is done in reduce phase - 1 node is assigned to each word. It receives count from machines for that word and sums them up.

Example2: 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.

Question: What is the reason to shuffle? Why cant you serialize it?
Answer: That is what we are doing. In the second phase, we have to decide who is responsible for which set of data and send it to that node. Because all the data incoming to the first phase (Node 1) in the case of sorting a data is random. So from second phase we decide what set of data will be handled by which set of nodes and forward it.

24.7.3 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

Question : How is HBASE using MapReduce?
Answer : HBASE is just a storage layer. MapReduce reads from the storage layer, in this case - HBASE

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.

24.7.4 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.

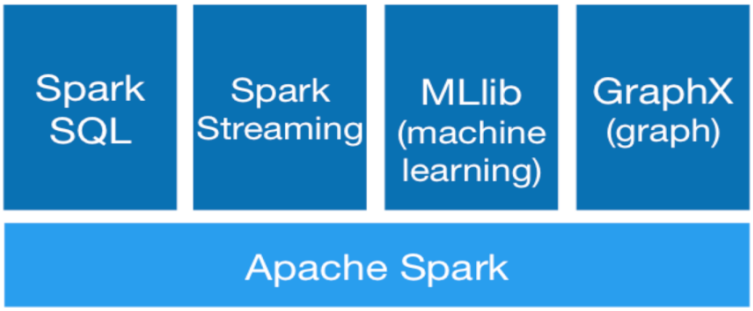


Figure 24.5: Spark Platform

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.

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 (Resilient Distributed Dataset). 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. If the data is larger than the memory available, data can spill over to disk.

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.

Question : Is Spark a type of distributed middleware?
Answer : Both Hadoop and Spark are type of distributed middleware designed for large data processing.

Question : In Spark where are the computations stored?
Answer : The computations are code that a user has written. Spark will keep track of graph as its generated. For each node it will keep track of what code was run to generate this output. We will only redo that part to do the minimum computation to generate the data.

Lecture 25:Distributed Systems Security

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 Security and Privacy

Security and privacy are related but different concepts when it comes to protecting data. Unauthorized access poses a security risk, potentially leading to the theft of sensitive information like credit card details or personal data.

However, privacy violations can happen even without a security breach. For example, a mobile app might collect location data for targeted ads, inadvertently revealing visits to a doctor. Similarly, seemingly anonymous browsing history can be used to uncover personal information. Even seemingly harmless data, like simple request patterns, can be used to learn internal details of complex systems, such as deep neural networks.

In summary, while security focuses on preventing unauthorized access and data breaches, privacy concerns extend to protecting personal information from exposure or misuse, even in the absence of a security breach.

25.1.2 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.

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.

25.1.3 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.

Question: Can there be a possibility where A and B are exchanging keys and somebody else can get the key? **Answer**: If you can't exchange keys securely, then there's a problem with the encryption algorithm, maybe the key is known to someone else. Will be discussed later in key exchange and how to make it secure. For now, we assume that key is not vulnerable.

Question: Do the nonces need to be truly random? **Answer**: No, they don't need to be random at all.

25.1.4 Authentication using public keys

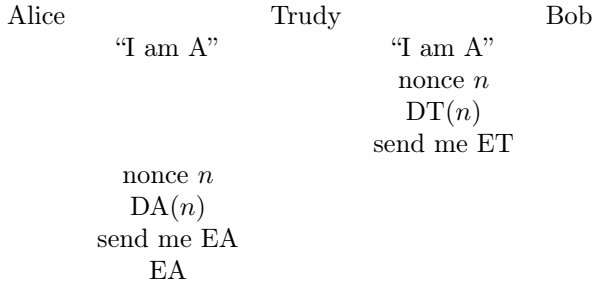
AP 4.0 uses symmetric keys for authenticantion. **Question**: can we use public keys? *symmetry*: DA(EA(n)) = EA(DA(n)).

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.

25.1.5 Man-in-the-middle attack

Trudy impersonates as Alice to Bob and as Bob to Alice.



Bob sends data using ET, and Trudy decrypts and forwards it using EA (Trudy *transparently* intercepts every message).

25.1.6 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?

25.1.7 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.

25.1.8 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.

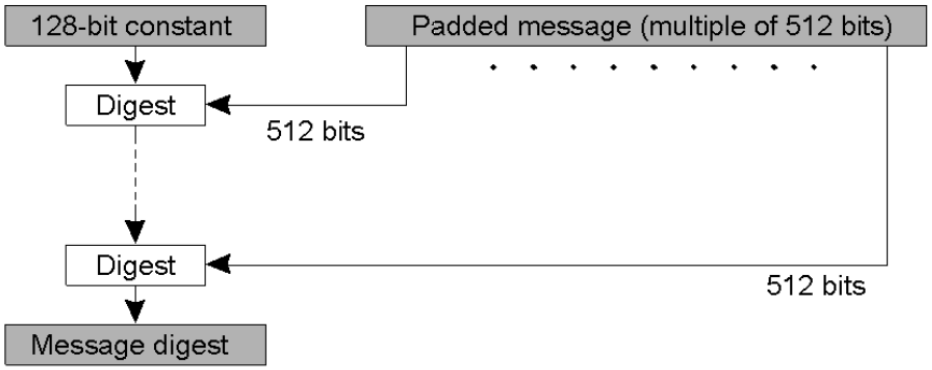


Figure 25.1: The structure of MD5.

25.1.9 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).
- 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.

25.1.10 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.

25.1.11 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.

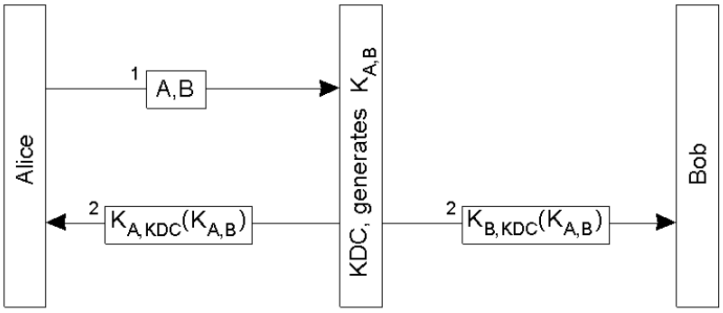


Figure 25.2: The principles of using a KDC.

Question: How did Alice and KDC generate that key securely in the first place? **Answer**: We assume that the original exchange is done in some secure way using physical exchange or something like that.

25.1.12 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.

25.1.13 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.

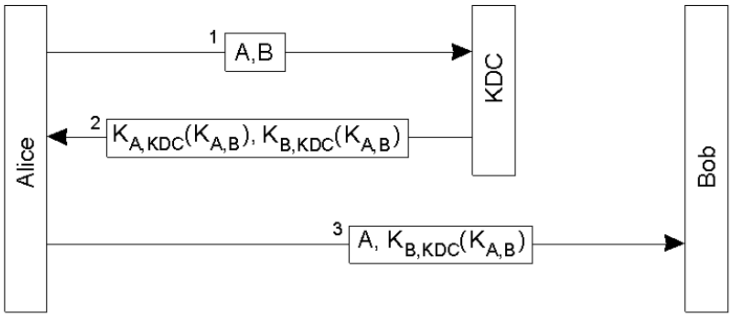


Figure 25.3: Using a ticket and letting Alice set up a connection to Bob.

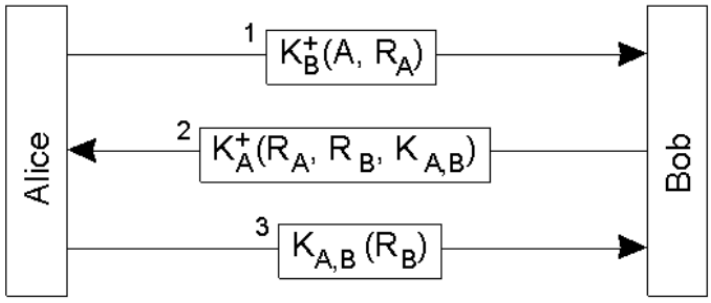


Figure 25.4: Mutual authentication in a public-key cryptosystem.

Question: What if we get a lot of encrypted or unencrypted messages that will increase load? **Answer**: You need things like firewall to filter out those packets. That's a different problem and these methods will not address that issue.

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 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.

25.1.14 Security in Enterprises

- In the multi-layered approach to security, Security functionality is spread across multiple components.
- Firewalls
- Deep packet inspection

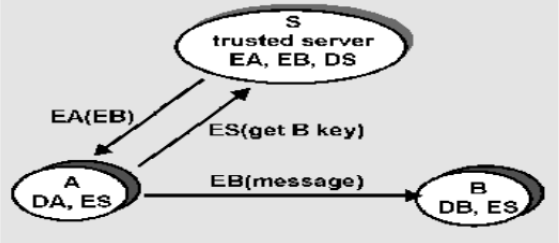


Figure 25.5: Public key exchange: trusted server.

- 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.

25.1.15 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

25.1.16 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

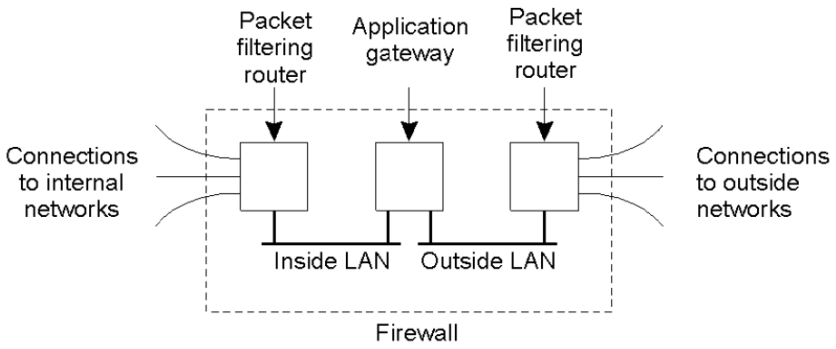


Figure 25.6: A common implementation of a firewall.

25.1.17 Access Control

Access control is a set of policies that determine which users can access specific resources. In addition to username and password, access control defines which users are permitted to access particular machines or resources. It is typically presented as a comprehensive table containing all relevant access information.

- Access control lists
- Capabilities
- Protection domains

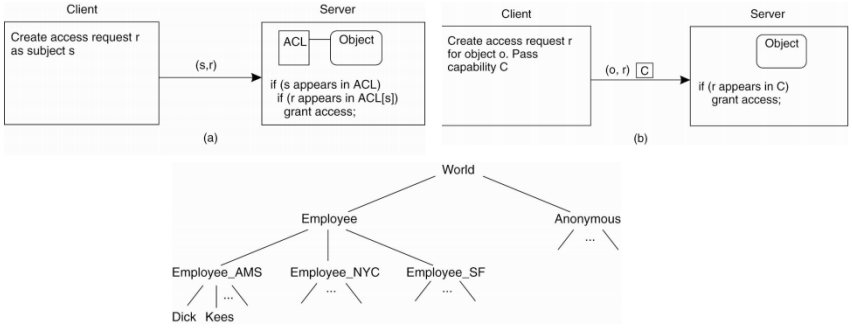


Figure 25.7: Access Control

25.1.18 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)

25.1.19 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.

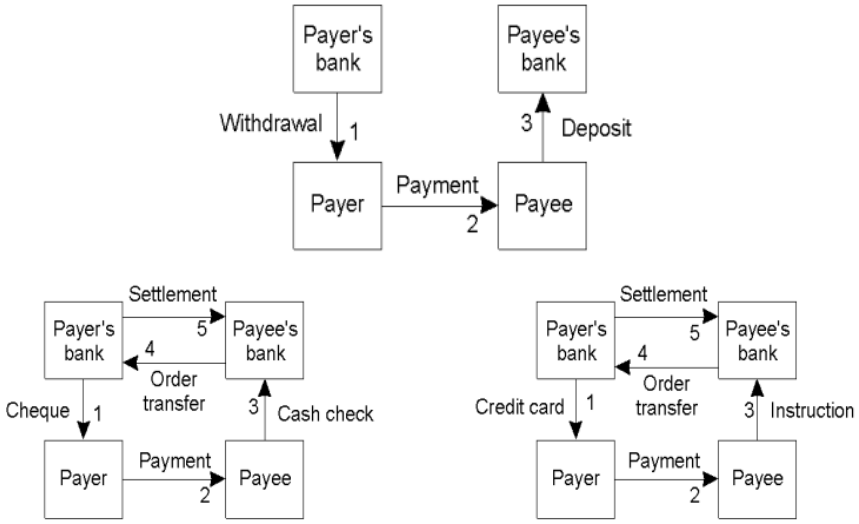


Figure 25.8: (a) Paying in cash; (b) Using a check; (c) Using a credit card.

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.

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.

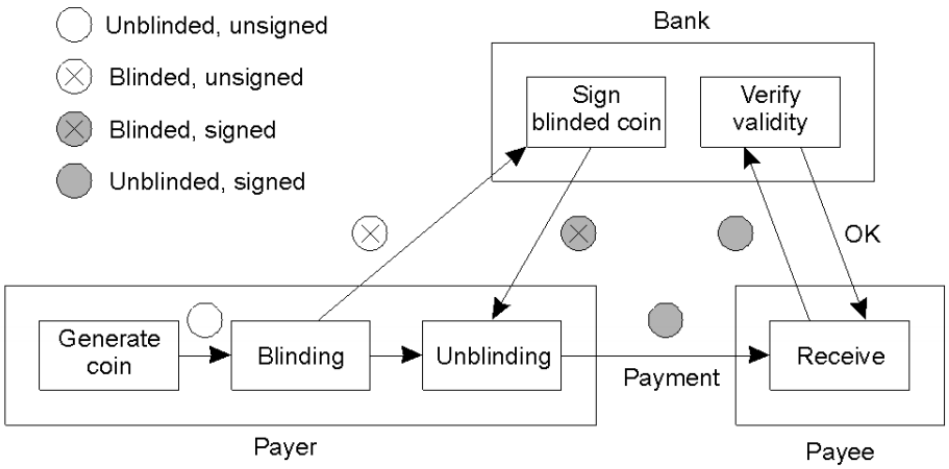


Figure 25.9: Illustration of E-cash.

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.

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.

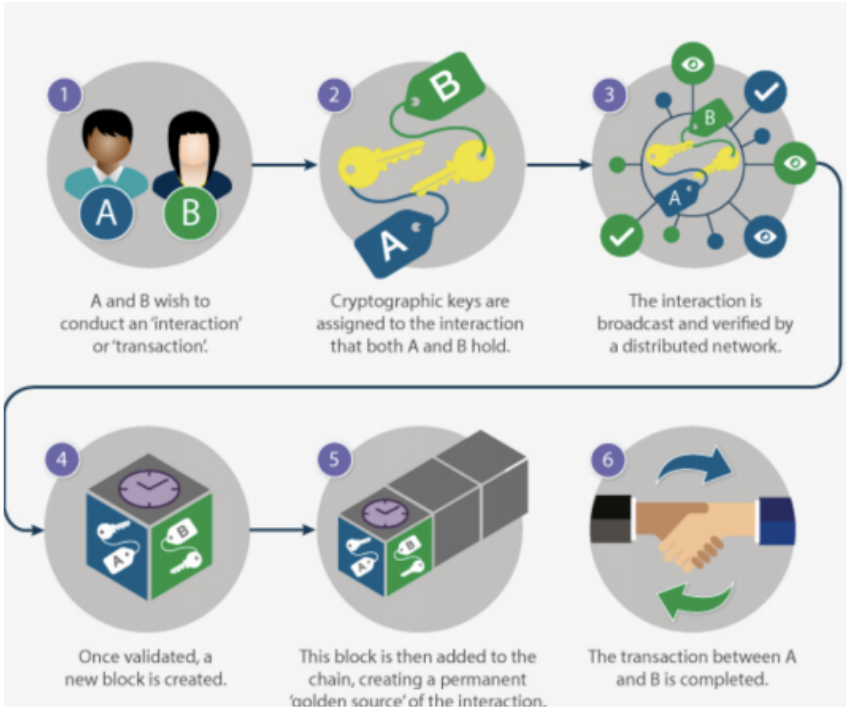


Figure 25.10: How Bitcoin works.

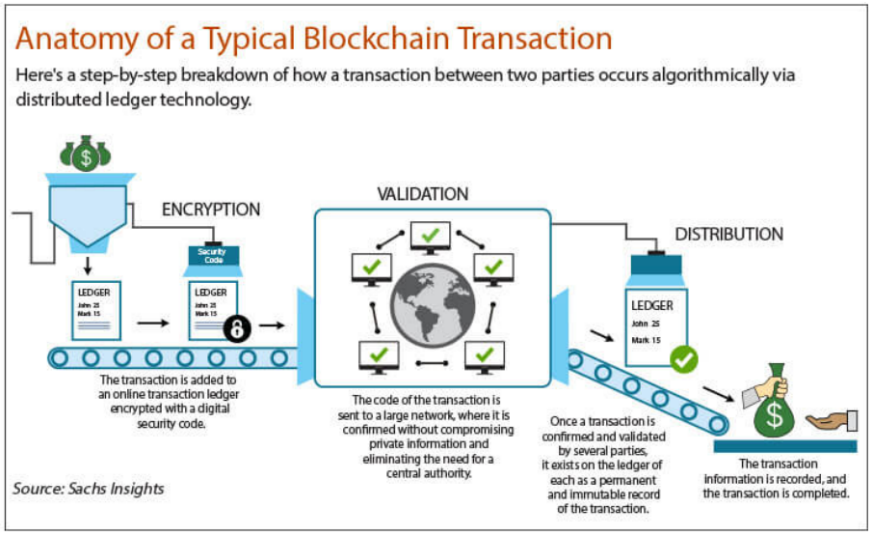


Figure 25.11: How blockchain works.