CMPSCI 677 Distributed and Operating Systems **Spring 2025**

Lecture 1: February 5

Lecturer: Prashant Shenoy
Scribe: Bin Wang(2018), Jonathan Westin(2019), Mehmet Savasci(2022), Thanathorn Sukprasert (2023)
Krishna Praneet Gudipaty (2024), Polina Petrova (2025)

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.

Grading: Two exams (one midterm and one final) (44%), three programming labs (43%), smaller assignments (homework and lablets) (12% total, roughly 1% per assignment), and class participation (in-class, Piazza, quizzes) (1%) will contribute to the final course grade. All programming work has to be completed in either Python or Java.

It is important to note the two pre-requisites for this class: undergraduate exposure to operating systems and good skills in a high-level programming language. It is your responsibility to keep these pre-requisites in mind, the professor will not check. Please note that **no laptop/device use is allowed during the class.**

Course tools:

- Piazza: discussions (post all questions here, emailing is discouraged)
- Gradescope: assignments and exams
- GitHub Classroom: labs

Important questions that were answered before the academic part was started:

1. Q: Is there a recommended textbook to review undergraduate OS? A: Professor will post a link to the freely-accessible online textbook on Piazza after class. You can also review the CS377 undergraduate lectures on the same YouTube channel for this class.
2. All three sections do the same work, same exams, and have the same grading policy.
3. 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 1: Can a decentralized system also act as a single coherent source?

Answer: Yes, decentralized systems are also distributed. Definition of distributed also applies to decentralized systems. Example of decentralized system: peer-to-peer system.

Question 2 (previous year): Is decentralized systems a subset of distributed systems?

Answer: Yes, decentralized systems are a subset of distributed systems. All decentralized systems distributed, but not all distributed systems are decentralized.

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 1 (previous year): 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 2 (previous year): 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 3 (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 4 (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 5 (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 6 (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 that is accessible by multiple clients. In this case, the processing capacity of the server will become a bottleneck when many clients try to access the single server machine. The solution is to replicate (distribute) 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). When the client sends a request to the server, the server queries the database and returns the answer. 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).

Note: it is not always necessary to distribute all components of the system. The architecture will depend on the kind of system you are creating.

Question 1: What is the difference between the service and the machine (server)?

Answer: The machine (server) is a piece of hardware. The service is a piece of software that runs on the server and provides a service that the clients can access.

Question 2: What are some examples of a decentralized algorithm?

Answer: Suppose you have a set of replicated servers (e.g. a large website replicated on multiple machines). When a request comes in, the service will decide which replica to send the request to. A centralized load balance algorithm will receive all the requests and then decide which server to send the request to. In this case, the algorithm can cause a bottleneck issue because the load balancer will need to keep track of the loads of each server. In a distributed version of the algorithm, a service or replica will receive the request directly and complete it, if able, or forward it to another server.

Question 3: Can the load balancer be considered as a service as opposed to an algorithm?

Answer: It depends, but typically you offer a load balancer as part of a larger service (e.g. web service) rather than offering it arbitrarily as a service. Therefore, it's considered an algorithm.

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 1 (previous year): 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 2 (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 3 (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 4 (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 5 (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, large clusters of servers run in facilities called data centers, which are 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 1 (by Instructor): What would we gain from a microkernel architecture from a monolithic architecture?

Answer (Student 1): It's less complicated to design an OS this way because you've modularized it and put each component in its separate process.

Answer (Student 2): It reduces the impact of failure. If the memory manager process fails, the OS won't stop functioning.

Question 1: Could all the processes be running parallel to each other (all at once)?

Answer: It depends on the hardware the microkernel is running on. A uniprocessor system won't have parallelism by definition, since there's one single CPU. But if the microkernel is running on a multiprocessor system, you will have parallelism.

Question 2 (by Instructor): What are some disadvantages of the microkernel architecture?

Answer (Student): You need to make multiple calls to process a request.

Answer (Instructor cont.): First, send the request to the microkernel, then the microkernel forwards the request to a module (e.g. memory manager), take action, and come back. A monolithic kernel will make a single system call that will execute code within the kernel and get a response.

Question 2: Is there direct memory access in a microkernel system?

Answer: We're not going to worry about that here because this is a general concept, not memory-specific. Any user application interactions with the kernel go to the microkernel first, then are sent to the user processes. They can't interact directly and have to go through the microkernel for security reasons, which slows the system down.

Question (by Instructor) (previous year): 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 1 (previous year): 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 2 (previous year): 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.