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

Lecture 1: January 26

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

1.1 Introduction to the course

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

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

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

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

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

1.2 Why should we learn about distributed systems?

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

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

Definition: Two definitions can be given.

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

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

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

Distributed systems also have several disadvantages:

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

1.4 Transparency in Distributed Systems

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

Here are some transparencies:

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

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

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

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

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

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

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

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

1.5 Open Distributed Systems

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

1.6 Scalability Problems and Techniques

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

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

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

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

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

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

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

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

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

1.7 Distributed Systems History and OS Models

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

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

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

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

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

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

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

Virtualization

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

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

1.8 Operating Systems History

1.8.1 Uniprocessor Operating Systems

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

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

1. **Monolithic model.** In this model one large kernel is used to handle everything. The examples of this model include MS-DOS and early UNIX.
2. **Layered design.** In this model the functionality is decomposed into N layers. Each layer can only interact with with layer that is directly above/below it.
3. **Microkernel architecture.** In this model the kernel is very small and only provides very basic services: inter-process communication and security. All other additional functionalities are implemented as standard processes in user-space.

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

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

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

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

1.8.2 Distributed Operating System

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

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