CMPSCI 677 Distributed and Operating Systems **Spring 2019**

Lecture 1: January 23

Lecturer: Prashant Shenoy *Scribe:* ***Jonathan Westin (2019), Bin Wang (2018)***

1.1 Introduction to the course

The lecture started by outlining the administrative stuff about the course web page (`http://lass.cs.umass.edu/~shenoy/courses/677`), course syllabus, course mail, staff, textbook, piazza, topics, and schedule. The instructor recommended "Distributed Systems, 3rd ed" by Tannenbaum and Van Steen as the textbook. However, some older material from the 2nd version of this book will also be used, so it's recommended to get them both.

The instructor also introduced the grading scheme, course mailing list, and other resources. It's worth noting that programming assignments and exams (1 mid-term and 1 final exam) contribute heavily towards the final grade. This course assumes undergraduate background in operating systems. No laptop/device use is allowed during the class.

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. 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. Therefore, it's useful to understand how these real-world systems work. This course will also cover the basic principles that underlie the design of these distributed systems.

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

Definition: A distributed system is a system which consists of multiple connected processors working together (usually on a collection of independent computers) and appears to its users as a single coherent system. The examples include parallel machines and networked machines.

Distributed systems have the following advantages: 1) 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) Distributed systems lead to better economics in terms of price and performance. It's 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's 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's usually hard to achieve linear scalability due to various bottlenecks (more in section **??**). 5) Potential for incremental growth. If an application becomes more popular, 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) Distributed applications are more complex in nature than centralized applications. They also require distribution aware programming languages (PLs) and operating systems (OSs) which are also more complex to design and implement. 2) 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 general design principle is that if an aspect of the system can be made transparent 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 result 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 server 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). However, although incorporating all the transparency features reduces complexity for users, it would also adds complexity for the system.

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

Answer: It depends, in general, it is cheaper to buy several machines to get the performance required but on the other hand this will give more hardware that can break or need maintenance. But in general we see that several machines often is cheaper than a super-computer.

Question (Prev year): Are there scenarios where you actually ought to reveal some of these features rather than making it 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 servers doesn't 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(Prev 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's 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
- centralized data
- centralized algorithms.

Centralized services simply mean 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 it will also make the system design more complicated.
Centralized data mean 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 mean that the algorithms used in the code make centralized assumptions (e.g. doing routing based on complete information). Generally speaking replication improves scalability but what aspects to replicate depends on the characteristics of the application.

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

1. No machine should have complete state information. In other words: No machine should know what happens on all machines at all times.
2. Algorithms should make decision based on local information instead of global information. For example, if you have a loadbalancer than forwards requests to the least loaded server, querying the load on every server every time a request comes will introduce a lot of overhead. A better design in this situation may be using only local information or using randomized algorithms.
3. Failure of any one component should not bring down the entire system. One part of an algorithm failing or one machine failing should not fail the whole application/system.
4. No assumptions should be made about a perfectly synchronized global clock. A global clock is useful in many situations (e.g. in a incremental build system) but you should not assume it's perfectly synchronized across all machines.

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

Question : What is an example of a make-decisions based on local and global information?

Answer: We will talk about distributed scheduling in a later lecture. As an example: 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 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 (Prev 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 Models

Minicomputer model : In this model each user has their 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 (LAN) that provides connection nearly all 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's clusters/data centers rather than individual servers that are interconnected over WAN.

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 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 that uses a one large kernel 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. Today this model is only used in the network sub-system. 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 (Instructor): Why is the microkernel architecture a good idea?

Answer (Student): Isolation, since it gives us security and modularity.

Answer (Instructor cont.): The microkernel architecture provides two advantages: 1) Modularity. It's easier to add new functionalities or modify existing functionalities. And if a bug occurs it doesn't effect other modules. 2) Security. Vulnerabilities in one module will not affect other modules. However, this architecture also has a major disadvantage: the performance may be not so good because of the overhead incurred by inter-process communication is much higher than function calls within a process. Therefore, most modern operating systems at one time employed the microkernel architecture but eventually have to move from it due to performance issues. Nowadays most operating systems use a hybrid architecture: some functionalities are independent processes while other functionalities are moved back to kernel for better performance.

1.9 Distributed Operating Systems

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 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. NOS provides very little transparency and the least form of integration. It's basically standard operating system kernel augmented with networking functionality so that communication with other machines is enabled. Besides that nothing more is hidden by the OS. Most modern operating systems are networked operating systems. Middleware are somewhere in between: the transparency is provided to only some applications while other applications still just use the regular OS networking capability.

1.10 Multiprocessor Operating systems

Multiprocessor operating systems are just like uniprocessor operating systems except for that they manage multiple CPUs/cores transparently to the user. Each processor has its own hardware cache so maintaining consistency between those caches becomes a challenge. Today most operating systems are multiprocessor operating systems because even mobile phone has multiple cores.