Lecture 3

12th January 2021

Team 4: Jayati Narang(2018101066), Virat Mishra(2019201033), Vishvesh Subramanian (20171133)

Presentation Scribe:

Topic: Applications and Challenges of Distributed Systems

Today, distributed computing is an integral part of both our digital work life and private life. Anyone who goes online and performs a google search is already using distributed computing. In the following sections, we present the various applications and the challenges associated with them.

Applications of distributed systems:

Distributed systems find their applicability in a host of practical scenarios mentioned below:

1. **Telecommunication:** The ad-hoc architecture based communication networks rely on the distributed communication responsibility of all the participating mobile nodes, wherein mobile nodes have to participate in routing by forwarding packets of other pairs of communicating nodes.

2. **Internet:** The World Wide Web (WWW) is a distributed system where there are multiple components under the hood that help browsers display content, but from a user's point of view, all they are doing is accessing the web via a medium (i.e., browsers).

3. **Wireless sensor networks:** A sensor network is an ensemble of nodes. Each node by itself is a tiny computer that may gather input data from its sensors and communicate with other nodes by radio. As radio signal strength deteriorates with distance quickly, it is generally infeasible to directly have all nodes talk to each other (or to a base station). Instead, nodes may have to

use other nodes as intermediate relays in order to communicate. This is where distributed algorithms orchestrate their communication and computation in order to solve a given task efficiently.

4. **Peer-to-peer applications**: Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. For example, in torrent file-sharing systems where all peers contribute as seeders in uploading the requested file.

5. **Multiplayer games:** The problem faced by client-server architecture in multiplayer games is the increase in load and traffic congestion on the server. One can overcome this problem using distributed architectures where peers can directly communicate (solving latency issues), and computational/storage capacities of individual nodes can be utilized, decreasing server dependency.

6. **Distributed database systems:** A distributed database is a database that is not limited to one system. It is spread over different sites, i.e., on multiple computers or over a network of computers. A distributed database system is located on various sites that do not share physical components. It will be required when a particular database needs to be accessed by various users globally. It needs to be managed such that for the users, it looks like one single database. Distributed algorithms need to tackle issues of concurrency, replication, latency, etc.

7. **Network File System:** Network File System (NFS) is a distributed file system (DFS) developed by Sun Microsystems. It allows directory structures to be spread over the networked computing systems. A DFS is a file system whose clients, servers, and storage devices are dispersed among the nodes of the distributed system. Hence, we need to manage these distributed systems in a way similar to that of distributed databases.

8. **Real-time systems:** A real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations but also on the time when the results are produced. Real-time systems are often implemented as distributed systems for fault tolerance, localized computations, and performance concerns.

9. **Grid computing:** Grid Computing can be defined as a network of computers working together to perform a task that would rather be difficult for a single machine. All machines on that network work under the same protocol to act as a virtual supercomputer. The task that they work on may include analyzing massive datasets or simulating situations that require high computing power. Computers on the network contribute resources like processing power and storage capacity to the network. Coordination of all these contributing nodes requires distributed algorithms.

Challenges associated with distributed systems:

The challenges concerned with the designing of distributed systems can be seen from either algorithmic or system-design point of view:

1. **Algorithmic challenges:**
    a. **Design of execution models/frameworks:** Models that define the sequence/order of events happening in a distributed system are crucial to the correctness of the distributed system, for instance, the partial order and interleaved model, which respectively specify the relative and absolute order of the events.
    b. **Concurrency:** Access to shared resources in distributed systems needs to be coordinated via mechanisms like mutual exclusion in scenarios involving database writes, using shared memory, etc. Deadlock detection should be coordinated to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.
    c. **The notion of Time:** In Centralized systems, there is no need for clock synchronization because, generally, there is only a single clock. In distributed systems, there is no global clock or shared memory. Each processor has its own internal clock and its own notion of time. In practice, these clocks can easily drift apart by several seconds per day, accumulating significant errors over time. Distributed systems are subject to timing uncertainties as certain processes may lack a common notion of real-time.
    d. **Debugging:** Debugging sequential programs on a uniprocessor/single machine environment is a reasonably well-understood task. Distributed systems, on the other hand, introduce enormous complications for debugging. The physical separation of processors and the communication delays between processors make halting all processors instantaneously impossible. Distributed systems require

debugging techniques that explicitly address the uncertainty inherent in distributed systems.

e. **Migration and load balancing:** Process Migration is the ability of a system (operating system or user-space program) to transfer processes between different nodes in a network. The motivations behind process migration are to balance the load, improve availability, enhance communication performance, and ease system administration. However, since a process involves and interacts with many different components of an operating system, migration of a process is technically complicated and demands a great deal of researches. Migration and relocation transparency need to be adhered to while designing algorithms for migration.
f. **Synchronization/coordination:** Coordination in a synchronous system with no failures is comparatively easy. However, if a system is asynchronous, meaning that messages may be delayed an indefinite amount of time, or failures may occur, coordination and agreement become more challenging. For example, like other broadcast networks, sensor networks face the challenge of agreeing on which nodes will send at any given time. Besides, many sensor network algorithms require that nodes elect coordinators that take on a server-like responsibility. Choosing these nodes is particularly challenging in sensor networks because of the battery constraints of the nodes.

2. System challenges

Following are the challenges associated with designing distributed systems:

a. **Communication:** Information exchange among various nodes is critical and is facilitated by mechanisms like remote procedure call (RPC), remote object invocation (ROI), message-oriented communication versus stream-oriented communication.
b. **Naming:** Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a transparent and scalable manner.
c. **Scalability and modularity:** Scalability is an essential indicator in distributed computing and parallel computing. It describes the system's ability to dynamically adjust its computing performance by changing available computing resources and scheduling methods. Modularity is the degree to which a system's components may be separated and recombined. A distributed system faces much variability in the functions that it needs to support and the

deployment environments in which it needs to execute. The modularity of functionality and composition is essential in achieving this variability.

d. **Transparency:** The concealment from the user of the separation of components of a distributed system so that the system is perceived as a whole. Transparency in distributed systems is applied in several aspects such as Access, Location, Migration, Replication, Concurrency, Failure, Parallelism.
e. **Replication and consistency:** When our system is replicated on n machines, we can use all of the resources available to those n machines to process requests. If one of the replicas in our distributed system goes down, we want at least one to still be up to server requests and ensure that data is still available. However, if one copy of a file is modified, we want all replicas to make the same update and remain consistent. This tends to be complicated and expensive.
f. **Fault tolerance:** Fault tolerance is the property that enables a system to continue operating correctly in the event of the failure of (or one or more faults within) some of its components. Unfortunately, the components of literally every system are naturally imperfect and, therefore, prone to failures that may render the system unable to provide the service. To tolerate the failure of some components, that is, to keep the service available despite these failures, the system must be equipped with redundancy in space and time. Fault-tolerant distributed computing refers to the algorithmic controlling of the distributed system's components to provide the desired service despite certain failures in the system by exploiting redundancy in space and time.
g. **Security:** Securing distributed systems is both harder and easier compared to securing monolithic systems: Harder, because there is no central point of control, and easier, because the distribution itself gives a system certain beneficial characteristics, especially considering the elimination of a single point of failure. The attackers usually practice the following attacks: unauthorized, unfriendly access to the information, manipulation, and falsification of information (in the messages or by their transmitters at the routes).

Questions asked based on presentation:

Q1: Why is load balancing an algorithm challenge, shouldn't it be a system challenge?

Ans: While dividing the traffic (load) to each server of the distributed system, there has to be a mechanism(algorithm) that decides the best traffic division to ensure no server is in an idle state when others suffer huge network traffic. Hence this division of work, known as load balancing, is an algorithmic challenge rather than a system challenge.

Q2: How is grid computing different from cluster computing?

Ans: **Cluster Computing:** A Computer Cluster is a local network of two or more homogenous computers.A computation process on such a computer network i.e. cluster is called Cluster Computing.

Grid Computing: Grid Computing can be defined as a network of homogenous or heterogenous computers working together over a long distance to perform a task that would rather be difficult for a single machine.

Difference between Cluster and Grid Computing:

Cluster Computing	Grid Computing
Nodes must be homogenous i.e. they should have same type of hardware and operating system.	Nodes may have different Operating systems and hardwares. Machines can be homogenous or heterogenous.
Computers in a cluster are dedicated to the same work and perform no other task.	Computers in a grid contribute their unused processing resources to the grid computing network.
Computers are located close to each other.	Computers may be located at a huge distance from one another.
Computers are connected in a centralized network topology.	Computers are connected in a distributed or decentralized network topology.
Whole system functions as a single system.	Every node is autonomous, and anyone can opt out anytime.

Q3: Can load balancing be achieved without specific servers incharge of balancing load?
Ans: Yes, load balancing can be achieved without a load balancer. You can refer to these articles to get a better understanding. Link 1, Link 2.

Lecture Scribe:

Scalar time represented only the idea of the node's own time. To get an idea of other processes' time, we use the concept of vector time.

Vector time:

- Time is represented by n-dimensional non-negative integer vectors.
- Each process p_i maintains a vector vt_i [1..n], where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i and $vt_i[j]$ tells p_i's latest knowledge of process p_j local time.

1. Update rules :
    a. Node should increment it's own time by some non zero value before executing any event.
    b. For other nodes' times, they should be updated whenever an update from another node is received, updated value being maximum of the current of that kth process and new value received for that kth value.

Q4: ***Is this some kind of broadcasting?***

Ans: No, since a process i is sending a message to process j, piggybacked with its understanding of time. It is not sending this message for all the processes. Hence it's not broadcasting.

Q5: ***Does vector time limit the system from adding the node dynamically?***

Ans: No, mostly we start with a set of nodes in which some nodes may fail over time; hence the size of the vector time array does not increase. However, yes, if we have to add a new node in a very rare case, then we have to restart that whole system and increase the vector time array size by 1.

Q6: ***For process i, for d=1, are the events internal to the process ordered as a sequence 1,2,3,4,.... Or could there be jumps ?***

Ans: No, there can't be jumps for any process i, since the timestamp for this process is increased only when an internal process occurs.

Q7: ***Using vector clocks, can determine whether the two events are congruent or whether they are causal ordered?***

Ans: Yes, the vector clock is strongly consistent. To check if two events are causal events, then for each index one vector should dominate the other vector. Here dominate means one value should be more than the other. For congruent events this condition will fail.

To state this mathematically: For events a and b with vector timestamps t_a and t_b:

Concurrent: $t_a \parallel t_b$ iff not($t_a < t_b$) and not($t_b < t_a$)

Q8: ***Given an event e has timestamp vh, vh[j] denotes the number of events executed by process p_j that causally precede e, what is the number of events that causally precede e in the distributed computation?***

Ans: $(\sum_j vh[j]) - 1$ represents the total number of events that causally precede e in the distributed computation, where j is [1, n], n is the total number of processes.

Q9: ***Can two events have the same vector clocks?***

Ans: No, since for the internal events for process i, the i^{th} index is constantly increasing, hence that index will be different for sure. For the other process, say j,when the message is passed, j's own vector clock's index is increased hence the vector clock won't be the same. Hence two vector clocks' across the processes won't match. Thus two events can't have the same vector clocks.

Limitations of vector time : Message size is large in vector time due to the vector time array, which is piggybacked on the message. Sometimes even when the process has stopped, we still have its space and value stored in the vector time array, which increases the message overhead. Usually, a large number of processes are involved in the system, which presents us with the difficulty of passing around those vectors every time there is an update.

Q10: ***What can we do to reduce this load?***

Ans: One observation that we were passing around the whole vectors, with all it's updated and non-updated entries, gives us the scope to optimize the approach here. Only updated entries should be communicated. But this requires us to store the last vector time we sent an update to every other process, requiring $O(n^2)$ storage per process and $O(n^3)$ storage overall.

Singhal- Kshemkalyani Differential Technique:

This technique solves the previous $O(n^2)$ storage problem by maintaining two additional vectors per process:

- $LS_i[1..n]$ ('Last Sent') : $Ls_i[j]$ indicates the value of $vt_i[i]$ when process p_i last sent a message to process p_j.
- $LU_i[1..n]$ ('Last Update'):$LU_i[j]$ indicates the value of $vt_i[i]$ when process p_i last updated the entry $vt_i[j]$.

$LU_i[j]$ needs to be updated only when the receipt of a message causes p_i to update entry $vt_i[j]$.

$LS_i[j]$ needs to be updated only when pi sends a message to p_j.

Therefore we will send only those elements for which $LS_i[j] < LU_i[k]$ holds; which essentially means that the update time of these entries is more recent than the time they were last sent. Thus by using Singhal- Kshemkalyani method, we require only O(n) storage per process and $O(n^2)$ storage overall.

Matrix time:

In scalar time, every process tries to maintain the notion of it's own time only. In vector time, it tries to keep track of time of all other processes as well. Now in matrix time, the process tries to see the time of all processes from the point of view of every other process.

$mt_i[j, k]$ represents the knowledge that process p_i has about the latest knowledge that p_k has about the local logical clock, $mt_i[i, i]$, of p_i . mt_i stands for matrix of i^{th} process.

1. Update rules:

1. Updating own time : $mt_i[i, i] := mt_i[i, i] + d$ $(d > 0)$
2. Updating other entries, when a message is sent
   (a) $1 \leq k \leq n : mt_i[i, k] := \max(mt_i[i, k], mt[j, k])$ (That is, update its row $mt_i[i,*]$ with the p_i's row in the received timestamp, mt.)
   (b) $1 \leq k, l \leq n : mt_i[k, l] := \max(mt_i[k, l], mt[k, l])$

Q11: ***When some process sends a vector then only two rows modifies one is receiving process row and other is sending process row?***

Ans: This is incorrect, when some process sends a vector, only the process who is receiving it is modifying its vector.

Q12: ***Why do we require matrix clocks?***

Ans: It is required since, at some points, we might want to know whom all have received particular messages or whom all have made progress till some point in time, so there are requirements in distributed systems where I might want to dispose of some stuff that I have been storing or holding onto when I get to know that everyone has received that particular message. Thus how do I know how much progress other people have made? One way is to look at my vector clock, but my vector clock is limited by the sends that have been received by me, so if I can also use all the vector clocks of other people, I might get more information.

Homework questions:

1) Is vector time strongly consistent? That is, $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$? If so, prove else provide a counterexample.
2) Think about how vector clocks work for out of order msgs. Start by looking at the case where every message is a broadcast message.
3) Will Singhal-Kshemkalyani technique work if the messages are not in FIFO order?
4) Can we do something similar to Singhal-Kshemkalyani technique for matrix clock for its reduction of space?