DSC204A: Scaleable Data Systems Winter 2025

Lecture 22: Introduction to Distributed Computing

Instructor: Umesh Bellur *Scribe: Wendi Tan*

1. What is a Distributed System?

A distributed system is a collection of independent computers that appear to its users as a single, coherent system. Leslie Lamport offered a more colloquial definition: "You know you have one when the failure of a computer you don't even know existed stops you from getting your work done".

2. Why Distributed Computing?

There are several compelling reasons for adopting distributed computing approaches:

- **Economics:** A collection of microprocessors generally offers a better price-to-performance ratio than traditional mainframes. This provides a cost-effective way to increase computing power.
- **Speed:** A distributed system can possess more total computing power than a single mainframe. For example, it's challenging to build a single 500,000 MIPS processor, but this can be achieved by distributing the load across many smaller CPUs. Performance can also be enhanced through load distributing.
- **Inherent Distribution:** Some applications are naturally distributed, such as a supermarket chain with multiple branches.
- **Reliability:** If one machine crashes, the system as a whole can continue to operate, leading to higher availability and improved overall reliability.
- **Incremental Growth:** Computing power can be added in small increments, allowing for modular expandability as needs grow.

3. Disadvantages of Distributed Computing

Despite the benefits, distributed computing presents certain disadvantages:

- **Complexity:** Managing many interconnected components is inherently complex.
- **Failures:** Failures of network links and compute nodes are more common in distributed environments.
- **Security:** A distributed system often has a greater threat surface, making security a more significant concern.

4. Key Characteristics

Distributed systems are defined by two fundamental characteristics:

- **Lack of shared memory:** Each computer in a distributed system has its own memory; there is no globally shared memory space.
- **Lack of a common clock:** There is no single, global clock that can perfectly synchronize events across all machines in the system.

5. Core Design Issues

The characteristics mentioned above lead to several critical design issues that must be addressed:

- **Transparency:** This refers to achieving a "single-system image," making a collection of computers appear as a single computer to the user.
- **Reliability:** Systems must be designed to handle failures of compute nodes as well as network links. For example, with 3 machines, each having a 0.95 probability of being up, the probability of the system (at least one machine) being up is $1 - (0.05)^3$.
- **Performance:** Strategies are needed to overcome the latency introduced by the network.
- **Scalability:** Systems must be designed to effectively scale to potentially thousands of nodes distributed across large networks.

6. Fundamental Challenges

Several inherent challenges arise when building and managing distributed systems:

6.1 No Global State

No single host possesses complete, up-to-date knowledge of the entire system's state. State information must be exchanged over the network, but network capacity is limited, meaning not all information can be sent. The information received might be incorrect or outdated because new information takes time to propagate, and other changes may occur in the meantime. A key issue is detecting and addressing these inconsistencies.

6.2 No Absolute Time

Time cannot be measured perfectly across a distributed system. Hosts have different clocks that can skew, and the network can delay or even duplicate messages. This makes it difficult to determine the precise order of events, such as which player shot first in a game or who bought the last seat in an airline reservation system. A more nuanced abstraction is needed to represent time.

6.3 Failures

In distributed systems, failure is a common occurrence, not an exception; as systems become more complex, the likelihood of failure increases. Systems must be designed to tolerate these failures. For example, in web systems, contingency plans are needed for server failures. Effective systems need mechanisms to detect failures and recover from them.

6.4 Scalability

Systems tend to grow over time, requiring designs that can handle future increases in users, hosts, and networks. For instance, in a multiplayer game where each user sends their location to all other users, the message complexity is $O(n^2)$, which can quickly overwhelm networks as the number of users (n) grows. Strategies to manage this include reducing update frequency (though this has implications) or being selective about which nodes update each other.

6.5 Security

Distributed systems often involve many different entities that may not be mutually trusting (e.g., participants in a stock market) or under centralized control (e.g., the World Wide Web). Economic incentives for abuse can also exist. Therefore, systems frequently need to provide:

- **Confidentiality:** Ensuring only intended parties can read information.
- **Integrity:** Guaranteeing that messages are authentic and haven't been tampered with.
- **Availability:** Making sure the system cannot be brought down by attacks.

6.6 Consistency under Replication

To achieve scale, distributed systems heavily leverage concurrency, such as using a cluster of replicated web servers or a swarm of downloaders in BitTorrent. This often leads to concurrent operations on a single data object. The challenge is to ensure that this object remains in a consistent state. For example, in a banking system, mechanisms must prevent an account from being overdrawn due to concurrent withdrawals. Solutions include serialization (making operations happen in a defined order), transactions (detecting and aborting conflicting operations), and append-only structures (dealing with conflicts later).

7. System Architectures

Two primary architectural models are common in distributed systems:

- **Client-Server:** The system is divided into clients (often with limited power and scope) and servers (typically more powerful, with greater system visibility). Clients send requests to servers, and servers respond.
- **Peer-to-Peer (P2P):** All hosts are essentially "equal," acting as both clients and servers. Peers send requests directly to each other. P2P systems can be more complicated to design but offer potentially higher resilience.

8. Timing Models

The assumptions made about timing in a system define its timing model:

- **Synchronous Systems:** These assume lockstep synchronism for both computation and message delivery across all links. Key assumptions include:
  - Each message is delivered within a bounded time.
  - Each step in a process takes place within a lower and upper time bound ($lb < t < ub$).
  - The drift of each local clock has a known bound.
  - Multiprocessor systems are an example of synchronous systems.
- **Asynchronous Systems:** These make no assumptions about bounded computation or message delivery times, and clock drift is also considered unbounded. Most real-world distributed systems fall into this category.

9. Network Topologies

The physical or logical arrangement of nodes and connections in a distributed system is its topology. Common examples include:

- **Rings:** Can be unidirectional or bi-directional.
- **General graphs:** Arbitrary connection patterns.
- **Fully connected vs. not fully connected:** Whether every node is connected directly to every other node.

10. Fault Models

How failures are characterized and anticipated is defined by the fault model:

- **Failstop:** A process fails by simply stopping and does not resume or behave erratically.
- **Byzantine (Malicious):** A process can exhibit arbitrary behavior, including sending incorrect or misleading messages, and thus cannot be trusted. This model has significant implications for designing failure detectors, especially when comparing synchronous versus asynchronous systems.

11. Complexity Measures

When analyzing distributed algorithms and solutions, two primary complexity measures are used:

- **Time complexity:** In a synchronous system, this often refers to the number of rounds required for an operation. For asynchronous systems, upper bounds are often driven by timeouts for analysis purposes.
- **Message complexity:** This considers the number of messages exchanged or the total number of bits transmitted.

12. Qualitative Measures of Distributed Solutions

Beyond quantitative complexity, certain qualitative properties are desirable:

- **Safety:** Ensures that bad things do not happen, e.g., two processes should not access shared resources simultaneously.
- **Liveness:** Ensures that good things eventually happen, e.g., every request for a critical section is eventually granted.
- **Fairness:** Ensures that requests are granted in an equitable manner, often in the order they are made.

13. Common Problems of Interest in Distributed Computing

Several canonical problems are frequently studied in the context of distributed systems:

- **Leader Election:** The process of designating a single process as the coordinator or leader. An example is leader election in a synchronous, unidirectional ring.
- **Mutual Exclusion:** Ensuring that only one process can access a shared resource or critical section at any given time.
- **Distributed Key-Value Stores:** Systems like CHORD that manage data storage across multiple nodes.
- **Replication and Consistency:** Maintaining multiple copies of data across different nodes while ensuring the data remains consistent. This is useful in Distributed File Systems (DFS) and distributed shared memory/KV stores.
- **Consensus:** The task of getting all processes in a group to agree on some value or sequence of operations, such as deciding the order of writes into a replicated log (e.g., using algorithms like RAFT).