What I have learned in Distributed Systems: Overview and Related Contents

Qi Xiang Zhang
Institute of Information Management,
National Yang Ming Chiao Tung University, Taiwan

***Abstract*—This document will primarily share the knowledge the author has learned in the course of distributed systems in a simple manner. We will not delve too deeply into the underlying technical principles of distributed systems, and we hope that readers will be able to gain a general understanding of distributed systems without any prior knowledge when reading this document.**

***Index Terms*—Distributed Systems.**

I. Introduction

In a distributed system, each node is an element with autonomous computing capability. When referring to a distributed system, if there is no process of collaboration among nodes, it cannot be considered a distributed system. In a distributed system, each node has its own notion of time. Therefore, in the absence of a global clock and when each node has its own system clock, how to synchronize the various nodes becomes a key issue that distributed systems need to address. There are many other problems that distributed systems need to solve, such as how to manage group membership, how to ensure that a group of nodes is indeed communicating with authorized nodes, and whether it is necessary to store the survival state of nodes during the process. Additionally, when designing distributed systems, it is also necessary to consider whether the default condition is that nodes can freely enter and exit or whether there are restrictions. Finally, there's one more issue that must be mentioned: how to ensure that the required data can be found in a dynamic network structure when the relationships between nodes and their adjacent nodes are dynamic.

Next, let's discuss overlay types in distributed systems, with peer-to-peer (P2P) networks being one of them. However, we generally classify distributed systems into structured and unstructured types. In structured distributed systems, there exists a group of well-defined neighbors. However, in the case of mobile networks, ensuring the stability of neighbors within the group is a key issue. In unstructured distributed systems, there are no predefined neighbor groups, and nodes are randomly selected, with no structured network topology. This can result in the need to search through many nodes when searching for data. However, in distributed systems, neither structured nor unstructured is inherently better. We still need to decide whether to adopt a structured or unstructured distributed system architecture based on the current contextual requirements.

The next topic we'll address is coherent systems, which refer to a collection of nodes operating as a whole, where end-users do not need to know where computations are taking place. However, there are some inevitable issues within coherent systems, including partial failures, failures of parts of the distributed system, difficulty in recovery, and determining the proportion of nodes required to maintain the normal operation of the distributed system, among others. Middleware is also a crucial component of distributed systems, acting as the operating system (OS) of the distributed systems. After discussing the general concept of distributed systems, we need to determine what objectives distributed systems can help achieve, such as supporting resource sharing, distribution transparency (including access, location, relocation, migration, replication, concurrency, and failure handling), openness, and scalability, among others.

In distributed systems, the degree of transparency is also a crucial issue that needs to be discussed. Aiming for full distribution transparency may be overly ambitious because completely hiding errors in distributed systems is nearly impossible. Additionally, it's not appropriate to attribute slow computational speed to failures in distributed systems. Therefore, deciding how often to update information between nodes in the distributed system is important. However, pursuing full transparency may significantly impact the overall performance of the distributed system. On the other hand, exposing distribution can be beneficial, especially for applications like location-based services. The openness of distributed systems ensures that nodes can communicate in different execution environments. But achieving this requires distributed systems to have well-defined interfaces. Determining the policies or mechanisms for these interfaces, such as dynamic cache policies or various encryption methods, is another topic worthy of discussion.

The next topic to discuss is scaling in distributed systems, which includes size scalability, geographical scalability, and administrative scalability, among others. However, in practice, discussions often revolve around size scalability. In geographical scalability, the issue of latency in communication between nodes is prevalent. On the other hand, administrative scalability often involves computational grids. Techniques for scaling in distributed systems include asynchronous communication, separate handlers for incoming responses, moving computations to clients (where clients can assist servers with some preliminary work, such as validating form data on the client side), mirrored websites, web caches, file caching, and more. However, scaling in distributed systems also encounters issues with replication. As the scale increases, ensuring that changes made to a single node can quickly synchronize across the entire distributed system, while also ensuring consistency and achieving global synchronization, becomes challenging. Non-technical issues are often the most difficult to resolve in this context.

Furthermore, when delving deeper into the discussion, it's important to address some pitfalls encountered in the development of distributed systems. When designing distributed systems, we often fall into the trap of false assumptions and unnecessarily overcomplicate the design. False assumptions in distributed systems include assuming that the network is trustworthy, secure, and homogeneous, as well as assuming that the network topology is fixed and immutable. Other false assumptions include expecting zero latency, infinite bandwidth, disregarding data transmission costs, and assuming that someone is solely responsible for overseeing the operation of the entire distributed system. These false assumptions can lead to inadequate designs and may result in systems that are not robust or scalable in real-world scenarios. Therefore, it's crucial to be aware of these pitfalls and design distributed systems with a more realistic understanding of the challenges and constraints they face.

The operation modes of distributed systems can be categorized into parallel computing, grid computing, and cloud computing, with a focus on discussing the latter two. In grid computing, it assumes the presence of virtual organizations and allows nodes for collaborations. The structure of grid computing can be divided into four layers: the fabric layer, connectivity layer, resource layer, and collective layer, with the applications layer being the fourth. Additionally, cloud computing can also be segmented into four layers: the hardware layer, infrastructure (IaaS), platform layer (PaaS), and application layer. The advantage of cloud computing lies in its ability to assist enterprises in integrating different applications, as the network applications used among different organizational units within the enterprise may vary, making integration more challenging.

In practical terms, integrating different applications involves various techniques and technologies. These applications may include file transfer, shared database access, remote procedure calls (RPC), messaging systems, and more. Distributed systems can leverage a variety of tools and technologies to provide communication facilities and integration. These include:

1) **Transaction Processing Monitors (TPM)**: TPMs manage and coordinate transactions across distributed systems, ensuring atomicity, consistency, isolation, and durability (ACID properties).
2) **Remote Procedure Calls (RPC)**: RPC mechanisms allow programs to execute procedures or functions on remote systems as if they were local, enabling seamless interaction between distributed components.
3) **Parallel Message Interface (PMI)**: PMI facilitates communication and coordination between parallel processes or nodes in distributed computing environments, often used in high-performance computing clusters.
4) **Message-Oriented Middleware (MOM)**: MOM systems enable asynchronous communication between distributed components by facilitating the exchange of messages, ensuring reliable delivery and decoupling sender and receiver systems.
5) **Middleware**: Middleware serves as an intermediary software layer that enables communication and integration between disparate systems and applications, providing services such as message queuing, data transformation, and protocol mediation.
6) **Enterprise Application Integration (EAI)**: EAI solutions facilitate the integration of various enterprise applications and systems, enabling seamless data flow and business process automation across the organization.

By utilizing these communication facilities and integration techniques, distributed systems can effectively bridge the gap between different applications and enable seamless interoperability in complex computing environments.

Distributed pervasive systems have three subtypes, where the system assumes nodes are small and movable. In ubiquitous systems, the core elements are assumed to possess autonomy and context awareness, among other features. In the case of mobile computing, where numerous mobile devices act as nodes in the network, a key aspect is discovery. In situations where stable routes cannot be guaranteed, communication may become more challenging. The basic idea behind mobility patterns is pocket-switched networks, where a successful strategy involves distinguishing between friends and strangers. Nodes in motion can broadcast messages to their friend networks, and the message is then relayed by the first encountered friend to reach the target (first encounter).

In sensor networks, encouraging sensors to participate in network operations is a crucial issue. Nodes in the network may have varying computational capabilities, posing challenges. Sensor networks can be applied in distributed databases, but issues such as wasted network resources, resource allocation, and energy consumption persist. Despite having aggregation capabilities, sensors may still require returning too much data to the operator, presenting a challenge in terms of data management and efficiency.

II. Architecture

In the layered architecture of distributed systems (similar to OSI), there are three main types: pure layered organization, mixed layered organization, and layered organization with upcalls. An example of a layered organization with upcalls is a communication protocol, such as the communication (socket) between a client and server. In terms of application layering, it can be divided into three layers: the application-interface layer, processing layer, and data layer. As for System Architecture, there are different models. The centralized system architecture, for instance, is the basic client-server model. Additionally, the multi-tiered centralized system architecture includes single-tiered, two-tiered, and three-tiered architectures. As client computational capabilities strengthen, servers can offload some processing tasks to clients, thereby reducing the computational burden on the server.

Next, alternative organizations can be primarily categorized into three parts: vertical distribution (machines with different workloads), peer-to-peer, and horizontal distribution (machines with similar workloads). In structured peer-to-peer (semantic-free index), hash functions are commonly utilized. Examples include hypercube (suitable for scenarios with relatively stable nodes) and Chord, which involves the process of transforming to a ring (shortcut links). Typically, m-bit identifiers and m-bit keys are used, with nodes initially sorted in order. If a match is not found, there's no need to continue searching. In unstructured peer-to-peer networks, nodes can freely enter and exit the structure. Generally, the overall network structure is a random graph. Searching is often conducted through flooding (time-to-live) or random walk. Hierarchically organized peer-to-peer networks can be applied in Content Delivery Networks (CDNs), where super peers and peers can be distinguished. In super-peer networks, performance can be enhanced through index servers, and data storage can be more efficiently managed through brokers. One famous practical example is Skype. In edge-server architecture, collaboration is emphasized. A notable example is BitTorrent (torrent file, tracker, swarm). Under the hood, neighbor sets are regularly updated by the tracker, and when exchanging blocks, the file is divided into pieces.

The following content is additional supplementary material, primarily derived from Chapter 26, "Multithreading," in "Java How to Program." In multithreading, threads may not execute simultaneously but instead utilize rapid switching to create the appearance of simultaneous execution. Additionally, threads have different states in their life cycle (thread scheduling), including new, runnable, waiting, terminated, and blocked. Each Java thread also has a priority, determining its precedence in execution. However, threads may encounter synchronization issues. Generally, these are addressed using locks, such as the monitor's lock, to prevent multiple updates from occurring simultaneously. Asynchronous data sharing can result in concurrent writes to the same location, causing previous modifications to be overwritten by subsequent ones. In Java, ArrayBlockingQueue or BlockingQueue's put and take methods are commonly used to prevent such occurrences. Additionally, thread operation can be managed using wait, notify, and notify all methods, while Java provides lock and condition interfaces for synchronization purposes.

III. Concurrent and Distributed Systems

In this paragraph, we first discuss why making a system distributed is advantageous. The benefits of a distributed system primarily include:

1) **Inherently distributed**: A distributed system connects many nodes, distributing workload and resources across multiple machines.
2) **Better reliability**: Node failures do not disrupt the entire system's operation, enhancing fault tolerance.
3) **Better performance**: Data can be retrieved from nearby nodes, reducing latency and improving overall system performance.
4) **Solving bigger problems**: Distributed systems can parallelize the processing of complex tasks, enabling the solution of larger-scale problems by leveraging multiple nodes simultaneously.

Next, we address the reasons why one might opt not to make a system distributed. The drawbacks of distributed systems include:

1) **Communication failures**: Inter-node communication may fail, leading to disruptions in data exchange and coordination.
2) **Difficulty in diagnosing crashes**: Identifying which node has crashed can be challenging, complicating fault detection and recovery.
3) **Non-deterministic problem occurrences**: All problems may occur non-deterministically, making it challenging to guarantee fault tolerance and system stability.

The next topic to discuss is the concept of "hard drives in a van," where all data is stored in the same physical location. However, this approach poses some challenges, including high latency and high bandwidth requirements. Latency and bandwidth calculations are based on the time until a message arrives. For instance, if data is stored in the same building or central location, the latency is approximately 1 millisecond. If the data is stored on one continent and accessed from another, the latency can be around 100 milliseconds. However, if the data is stored on hard drives in a van, the latency can be as high as 1 day. Next, let's discuss Remote Procedure Call (RPC) and Service-oriented Architecture (SOA), both of which were mentioned earlier. In SOA, large software applications can be divided into numerous services, each serving a specific function. These services can be developed using different programming languages and communicate with each other via RPC, enabling modular and flexible software design.

The Two Generals problem is a classic issue in distributed systems, stemming from the inherent uncertainty present in such systems. It revolves around the question of how a general should decide whether to attack a city in the absence of complete information. One approach is to send a large number of messengers to increase the likelihood that at least one will successfully deliver the message to the other general. Another approach is for the general to attack only upon receiving a response from the other general, and likewise, the other general would only attack upon receiving a response. The main challenge in the Two Generals problem is the lack of common knowledge between the generals, leading to uncertainty and potential deadlock.

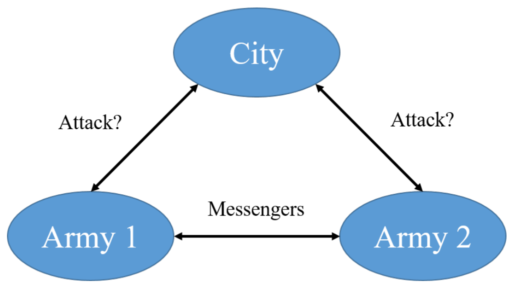

Fig. 1: The two generals problem

Expanding upon the Two Generals problem leads us to the Byzantine Generals problem, where we cannot guarantee that all generals are trustworthy. Messages may be tampered with during transmission, making it difficult to pinpoint which part of the distributed system is faulty (identifying malicious actors is challenging). In the Byzantine Generals problem, several characteristics emerge:

1) Honest generals do not know who is malicious.
2) Up to f generals might behave maliciously.
3) Generally, there needs to be a total of 3f+1 generals to tolerate the presence of f malicious generals.
4) Malicious generals may collude.

Cryptography may offer some assistance in addressing the problem, but it can also complicate matters further.

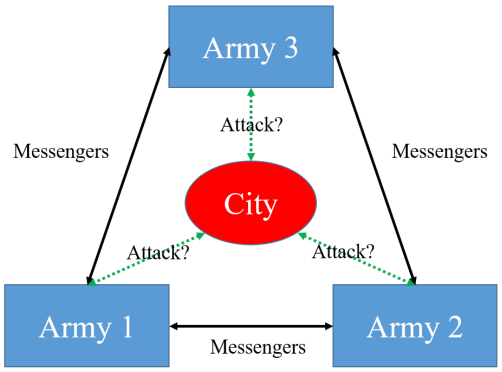

Fig. 2: The Byzantine Generals problem

In the system model, the Two Generals problem serves as a model of networks, while the Byzantine Generals problem represents a model of node behavior. Assumptions in a system model need to be captured, including:Network behavior: This encompasses message loss and can involve different types of network links:

- **Reliable links**: Messages are guaranteed to be delivered.
- **Fair-loss links**: Messages may be lost but not maliciously.
- **Arbitrary links**: Messages may be intentionally tampered with by an active adversary.

Network partition refers to situations where some network links drop or delay all messages for an extended period, resulting in two parties being unable to communicate with each other at certain times. Node behavior: Each node can execute a specified algorithm, but there are potential issues such as:

- **Crash-stop (fail-stop)**: Nodes can abruptly stop functioning and fail to respond.
- **Crash-recovery (fail-recovery)**: Nodes can fail but later recover and resume operation.
- **Byzantine (fail-arbitrary)**: Nodes can exhibit arbitrary and possibly malicious behavior.

These considerations are crucial for designing and analyzing distributed systems, as they impact system reliability, fault tolerance, and overall performance.

In terms of timing assumptions (synchrony), there are three main categories:

- **Synchronous**: This ensures that nodes can execute within a certain timeframe.
- **Partially synchronous**: There is some uncertainty in timing, but it is generally manageable.
- **Asynchronous**: Messages may experience random delays.

Regarding variations of synchrony in practice, the network typically exhibits predictable latency, but there are factors such as message loss requiring retries, congestion causing queuing, and network/route reconfiguration.

To summarize the content of the System Model:

- In the network aspect, considerations include reliable, fair-loss, and arbitrary behavior.
- In terms of nodes, considerations include crash-stop, crash recovery, and the Byzantine Generals problem.
- Timing considerations encompass synchronous, partially synchronous, and asynchronous behaviors.

The key takeaway is that before studying distributed systems, it's essential to understand the assumptions underlying the system model thoroughly. These assumptions play a critical role in the design, analysis, and performance of distributed systems.

In distributed systems, availability can generally be categorized into two types: Service-Level Objective (SLO) and Service-Level Agreement (SLA). Achieving high availability implies having a high degree of fault tolerance. In this context, ”failure” may render the entire system non-functional, while ”fault” may cause only a portion of the system to become non-functional. Faults can further be categorized into node faults and network faults. Fault tolerance ensures that the system can continue to operate even in the presence of faults.

As for failure detectors, there are two main types:

- **Failure detectors**: These algorithms detect whether another node is faulty or not.
- **Perfect failure detectors**: They label a node as faulty if and only if it has crashed, providing accurate fault detection.

These concepts and mechanisms are essential for ensuring the availability and reliability of distributed systems.

Next, let's discuss time, clocks, and the ordering of events. Firstly, we need to address how to handle time discrepancies between different nodes. In distributed systems, we often need to measure time and utilize scheduling, timeouts, failure detectors, retry timers, performance measurements, statistical profiling (logging information), log files/databases (recording when events occur), and data with time-limited validity (such as cache entries). Then, we need to determine the order of events across multiple nodes.

Regarding internal clocks within nodes, there are two types: physical clocks and logical clocks. In distributed systems, clocks do not simply measure time in seconds but determine the order of events based on their occurrence. Speaking of physical clocks, we cannot overlook the Quartz clock. Its operation principle involves counting the number of cycles to measure elapsed time. However, in distributed systems where precise timing is crucial, the Quartz clock may not be the best tool due to its susceptibility to various factors like temperature.

One common issue with Quartz clocks is drift, where one clock runs slightly fast while the other runs slow.

In practice, more precise timing tools like atomic clocks are commonly used. Other timing tools include GPS as a time source, which calculates position from the speed-of-light delay between satellites and the ground. Another timing tool is Coordinated Universal Time (UTC), but it faces the issue of the Earth's rotational speed not being constant. All these timing tools also encounter the issue of leap seconds, where there may be an extra or missing second in timekeeping. As for how computers represent timestamps, there are standards such as Unix time and ISO 8601. When faced with leap seconds, most software implementations typically choose to ignore them.

Livelock and deadlock are two important concepts in distributed systems. Livelock occurs when a program keeps switching between several states in a loop, unable to make progress (similar to two pedestrians continuously yielding to each other but both choosing to step back at the same time), while deadlock happens when processes are waiting for each other to release a resource lock, resulting in a standstill. Clock synchronization is affected by the issue of clock drift, where clock skew represents the difference between two clocks at a specific point in time. Two common protocols used in clock synchronization are the Network Time Protocol (NTP) and the Precision Time Protocol (PTP). Regarding atomic and time-of-day clocks, a time-of-day clock starts counting from a fixed point in time (e.g., January 1, 1970), and timestamps can be compared across nodes. On the other hand, monotonic clocks measure time relative to an arbitrary starting point.

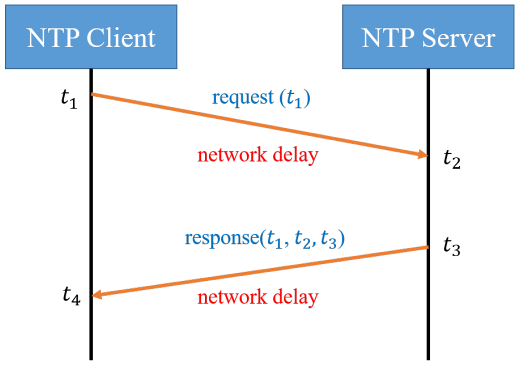

Fig. 3: Estimating time over a network (The round-trip network delay is equivalent to (t4 - t1) - (t3 - t2), but in practice, we don't know the time spent on the request and response.)

In the final part of this document, we will delve deeper into broadcast protocols and logical time. In practical distributed systems, it's challenging to define the true passage of time (physical timestamps inconsistent with causality), as even with synced clocks, t2 ¡ t1 is possible, but the timestamp order is inconsistent with the expected order. Next, we'll explain the difference between logical clocks and physical clocks. Physical clocks count the number of seconds elapsed, while logical clocks count the number of events occurred. Although physical timestamps are useful in various application scenarios, they may be inconsistent with causality. Therefore, in practice, we use logical clocks (which don't have a direct relationship to physical time) to capture causal dependencies. We will then introduce two different types of logical clocks, namely Lamport clocks and Vector clocks.

The Lamport clocks algorithm was first proposed in 1978. In this algorithm, each node initializes its local variable t to 0. Whenever an event occurs at a node, its local variable t is incremented by 1. Then, the variable t is transmitted over the underlying network link in the form of (t,m), where m is the message associated with the event. Upon receiving the transmitted t value, other nodes compare it with their own local variable t and take the maximum value. Subsequently, they increment their local t by 1 and continue to propagate it in the form of (t,m). However, the Lamport clocks algorithm suffers from a significant limitation: L(a)<L(b) does not necessarily imply that event a occurred before event b. Event a and event b could occur simultaneously or event a could precede event b. Both cases are possible. Below is a simple example of Lamport clocks:

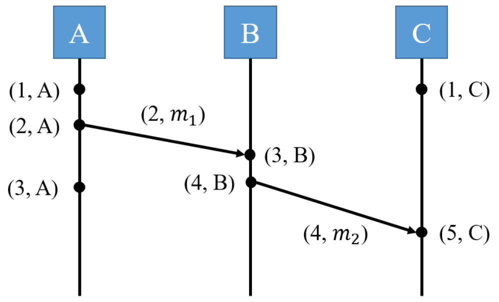

Fig. 4: Lamport clocks example

In Vector clocks, we assume there are a total of n nodes in the system, and we use V(a) to represent the Vector timestamp of event a, where t_i represents the number of events observed by node N_i. Each node has a current vector timestamp variable T (each node has a separate counter for every node for the number of events occurred). When an event occurs at node N_i, its variable T_i is incremented by 1. Subsequently, the current vector timestamp is attached to each message, and the receiving node merges the message vector into its local vector. Below is a simple example of Vector clocks:

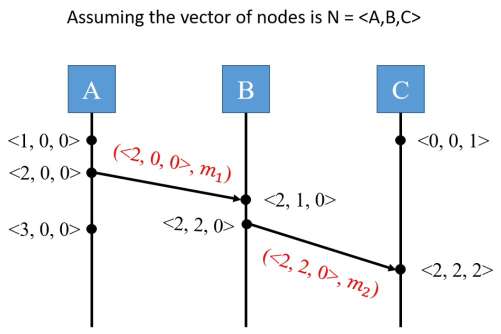

Fig. 5: Vector clocks example

After explaining the two different types of logical clocks, the next topic we are going to cover is broadcast protocols.

Broadcast protocols (multicast) are a form of group communication, where when one node sends a message, all nodes in this group will receive this message. The set of group members may be fixed (static) or dynamic, and if a node disappears, other nodes can immediately compensate. Broadcast protocols are more general than IP multicast. The system model constructed by broadcast protocols can be best effort (may drop messages) or reliable (non-faulty nodes deliver every message, by retransmitting dropped messages). Alternatively, the system model can belong to an asynchronous/partially synchronous timing model (with no upper bound on message latency).

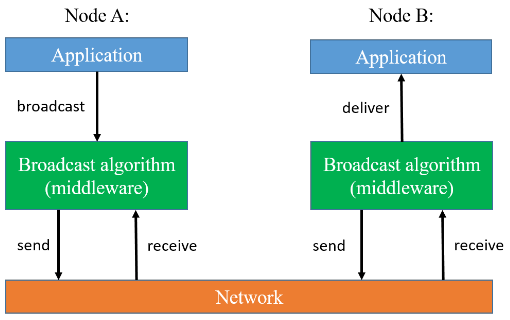

Fig. 6: Receiving versus delivering

The reliable broadcast comes in various forms, including FIFO broadcast, Causal broadcast, Total order broadcast, and FIFO-total order broadcast.In FIFO broadcast, it is assumed that m_1 and m_2 are broadcast by the same node. In this scenario, m_1 must be delivered before m_2, but there is no guarantee about the order in which they might arrive. Next, Casual broadcast is similar to FIFO broadcast. Here, m_1 must also be delivered before m_2. Following that is Total order broadcast. If m_1 is delivered before m_2 at one node, then m_1 must be delivered before m_2 at all nodes (everyone delivers some order, but may not send order). Finally, FIFO-total order broadcast combines FIFO broadcast and total order broadcast.

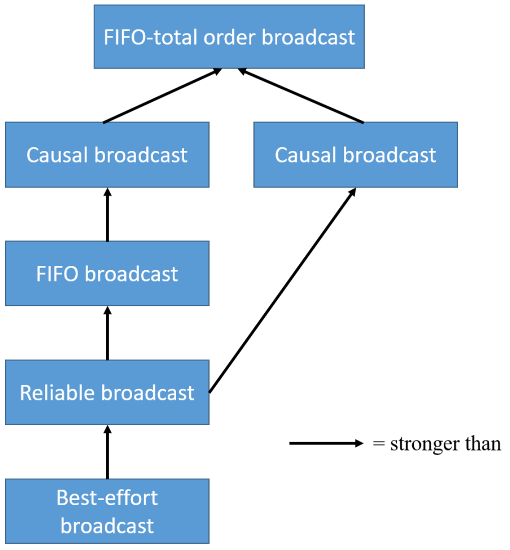

Fig. 7: Relationships between broadcast models

In addition to Broadcast protocols, this document will also introduce Gossip protocols, which are highly useful when broadcasting to a large number of nodes. The core idea is that when a node receives a message for the first time, it then forwards it to 3 other nodes chosen randomly. In simple terms, when a node receives data, it disseminates it to a specific number of other nodes, allowing other nodes to have backup data. However, Gossip protocols still have some issues to address. For instance, how to improve the success rate of searches when malicious nodes exist in the network, and how to ensure that all nodes successfully receive the message with the fewest hops possible.

IV. Conclusion

In conclusion, understanding the intricacies of distributed systems is essential for designing robust and efficient network structures. By delving into topics such as availability, fault tolerance, the Byzantine Generals Problem, logical clocks, scaling challenges, and common pitfalls, one can navigate the complexities of distributed systems with more clarity. It is crucial to address issues like false assumptions, replication challenges, and overcomplication in design to ensure the successful implementation of distributed systems. Embracing the nuances of distributed systems can lead to improved system performance, fault tolerance, and scalability in modern network architectures.

Acknowledgments

This document was originally written in Chinese and subsequently translated into English with the assistance of ChatGPT for translation and Grammarly for grammar correction. The content of the document is primarily based on the author's understanding of some topics learned in the Distributed Systems course offered by the Institute of Information Management at National Yang Ming Chiao Tung University. The document includes the author's own interpretation and assimilation of knowledge gained from the course; therefore, the accuracy of the content is not guaranteed. Furthermore, I would like to extend special thanks to the professor who taught this course, as well as to all the researchers who have contributed to the field of distributed systems. It is because of you that I have been able to delve into this fascinating domain.

Qi Xiang Zhang received the BBA degree from the Department of Information Management, National Central University of Taiwan (NCU), in 2023. He is currently pursuing the MS degree with the Institute of Information Management, National Yang Ming Chiao Tung University of Taiwan, NYCU. His main research interests include data mining, machine learning security and privacy, and network security.