Distributed Computing

Loic Niragire

School of Technology, Northcentral University

TIM 8121: Distributed Algorithms and Parallel Computing

Dr. Carol Cusano

May 7th, 2022

Distributed Computing

This is a technical report discussing current trends and techniques in distributed computing. First, a brief history of distributed computing is discussed, followed by a literature review of current trends and techniques. Then various architectures and current techniques for achieving resilient distributed systems are presented. The report concludes with a sample of applications leveraging distributed computation to deliver scalable solutions.

History

A distributed system is a collection of interconnected computing nodes that cooperate to achieve a goal by exchanging messages over a communication network, where a node could refer to a physical machine or a software process. Contrary to a multiprocessor system, nodes in a distributed system do not communicate through a shared memory. But rather, each node maintains a private memory, thus creating a distributed memory model. Thus, from a theoretic perspective, the theory of distributed computing is concerned with timing and topology uncertainties (Fraigniaud, Korman, & Peleg, 2013). For one, communicating nodes run at their own speeds and can possibly crash. Secondly, there is no knowledge of how far apart nodes are located.

The field of computational complexity theory has made significant contributions towards to the timing problem in distributed computing by establishing failure detection theories. Yet, determining whether a node has crashed or is running very slow remains unsolvable in asynchronous distributed systems. Thus, failure detectors only give hints about which nodes may have crashed (Chandra, Hadzilacos, & Toueg, 1992). Nevertheless, distributed systems aim to solve problems where each node only has a partial knowledge of the many parameters involved in the problem that needs to be solved (Raynal, 2018).

Communication

A distributed system is built on top of a communication medium that facilitates a reliable and secure data exchange between nodes. A closely coupled distributed system is traditionally characterized by a fast and reliable communication network where processors are physically close to one another. Whereas a loosely coupled distributed system refers to a system with physically dispersed processors with slow and unreliable communication (Bal, Steiner, & Tanenbaum, 1989).

Interprocess communication (IPC) is at the heart of distributed systems. To guarantee that a stream of bytes arrives without duplication or corruption, the TCP layer provides a reliable communication channel between two processes on top of the IP layer. The TCP protocol offers stability and reliability at the price of lower bandwidth and higher latencies than the underlying network can deliver. Alternatively, the User Datagram Protocol (UDP) is connectionless transport layer protocol without the reliability of TCP. Finally, the Transport Layer Security (TLS) encrypts data over the TCP channel.

Coordination

Once a secure and reliable communication is established between nodes, a distributed system relies on a coordinated effort between nodes to achieve the desired outcome. This coordination, for instance, determines the behavior of the system in the case of a failure detection. To facilitate coordination between nodes, the Domain Name System (DNS) allows nodes to discover each other over a network by resolving hostnames into IP addresses used to open TLS connections.

The possibility of node's failures further increases the coordination challenge of distribute systems. Thus, to reason about distribute systems, one must make a set of

assumptions about the behavior of nodes, communication links, and timing to verify the correctness of algorithms (Vitillo, 2021). There are three main models of communication links. The *fair-loss link* model assumes that messages may be lost and duplicated. Whereas the *reliable link* model assumes that a message is delivered exactly once, without loss or duplication. Lastly, the *authenticated reliable* link model makes the same assumptions as the reliable link but also assumes that the receiving node can authenticate the message's sender.

Different types of node failures can be modeled as either an *arbitrary-fault,* a *crash-recovery,* or a *crash-stop* model. The arbitrary fault model is also known as the Byzantine model. It assumes that a node can deviate from its algorithm in arbitrary ways. Meanwhile, the crash-recovery model assumes that a node can crash and restart at any time, but it does not deviate from its algorithm. On the other hand, the crash-stop model assumes that a node never comes back online once it crashes, but it does not deviate from its algorithm.

Three models are used to encapsulate a node's timing assumptions. The *synchronous* model assumes an upper bound time on sending a message or executing an operation. Whereas the *asynchronous* model assumes an unbounded amount of time for sending a message or executing an operation. Lastly, the partially synchronous model assumes that the system behaves synchronously most of the time, but occasionally may regress to an asynchronous mode.

Scalability

The stability of the coordination effort described above is profoundly reliant on the ability of the system to remain responsible under heavy loads. This is generally measured in terms of *throughput* and *response time.* A distributed system can increase its capacity either by scaling up or scaling out – the latter being more preferred. Scaling up a system involves

upgrading to more expensive hardware, whereas scaling out entails distributing the load over multiple nodes. There are three main architectural patterns used to scale out an application, functional decomposition, partitioning, and duplication.

Microservices dominate the current trend of decomposing a distributed system into its constituent parts. Resulting in a set of independently deployable services that communicate via well-defined APIs. A typical microservice architecture hides internal services behind an API gateway to enable service maintenance without constant client upgrades. Thus, the API gateway acts as the public API and provides features such as routing, composition, and translation.

Partitioning, or sharding, is another pattern used to scale out an application. Particularly, this technique is used when a dataset no longer fits on a single node. But it can also be used in other circumstances like sharding TCP connections. There are two common ways to achieve data partitioning, range partitioning or hash partitioning. Range partitioning splits data into partitions by key range in lexicographical order where each partition holds a continuous range of keys. Care must be taken to assert balanced partitions by using uniform distribution keys. On the other hand, hash partitioning ensures that the partitions are balanced by using a hash function to assign keys to partitions. Unlike range partitioning though, hash partitioning does not maintain the sort order over the partitions, requiring a secondary key to sort data within a partition (Vitillo, 2021).

Duplication offers horizontal scaling of stateless services by creating more instances. This technique requires a network load balancer to route service requests across a pool of servers. By decoupling the clients from the servers, the number of servers behind a load balancer can adjust transparently. Modern load balancers also offer other services such as

service discovery and health checks. Current trends support implementation of a load balancer at the DNS level, but flexible implementations operate at the TCP level of the network stack.

Duplicating stateful services behind a load balancer requires data replication across nodes. This becomes a challenge when dealing with dynamic data as it requires additional coordination to keep data in sync. The *single leader, multiple followers* approach is the most common technique for achieving data replication. Where a single node, elected as a leader, receives all write requests and is responsible for replicating the changes to the followers, synchronously or asynchronously.

Asynchronous replication model is not fault-tolerant as it replies to clients before the replication has been completed. Additionally, this model suffers from consistency issues since a client could request a read from a replica that has yet to receive the latest data update. Asynchronous replication is also known as an eventual consistency model. Meanwhile, synchronous replication comes with a performance penalty as data is first replicated to all followers before returning a response to the client.

A multi-leader replication system allows more than one leader node to accept write requests from a client. This is often preferred for applications with high write throughput, although it introduces a lot of complexity to resolve conflicting writes. Techniques relying on data structures with automatic conflict resolution, such as *conflict-free replicated data type* (CRDT), are often leveraged in multi-leader scenario. Lastly, data replication can be achieved without a leader node by replicas to accept write requests directly from clients. Although this strategy is more complex than the multi-leader replication.

Resiliency

Failure is inevitable as a distributed system scales out to handle more loads. This could result from an unreliable network, a slow process, unexpected load, or any of the many single points of failure in the system. To mitigate these failures, a risk score is typically calculated by multiplying the probability of a specific failure happening with its impact on the system. Therefore, by using collected risk scores, failures are prioritized and acted upon to increase a system's resiliency to failures. To maximize downstream resiliency, it is recommended to set timeouts when making network calls stop failures from cascading to the rest of the system (Vitillo, 2021). Additionally, clients need to implement retry mechanisms to handle timeout failures. On the other hand, a server's response time could increase significantly that it becomes unavailable to its clients. To avoid this case, services implement a load shedding technique to reject incoming requests when overloaded. Alternatively, *load leveling*, or *rate-limiting* techniques can be implemented to reject requests when specific conditions are met.

Applications

A distributed system is suitable for applications that require high availability and resilience to a single-node failures. For instance, Google drive and DropBox replicate data across multiple nodes to avoid data loss due to a single node failure. Applications handling heavy workload for a single node also make good candidates for a distributed architecture. These applications often have performance requirements that are physically impossible to achieve with a single node – Netflix for instance. More recently, blockchain technologies have highlighted distributed computing by implementing a decentralized ledger.

References

Bal, H. E., Steiner, J. G., & Tanenbaum, A. S. (1989). Programming Languages for Distributed Computing Systems. *ACM Computing Surveys, Vol. 21* (pp. 261-322). ACM.

Chandra, T., Hadzilacos, V., & Toueg, S. (1992). The Weakest Failure Detector for Solving Consensus. *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing* (pp. 147-158). Vancouver: ACM.

Fraigniaud, P., Korman, A., & Peleg, D. (2013). Towards a Complexity Theory for Local Distributed Computing. *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science (FOCS)* (pp. 35-61). New York: ACM.

Raynal, M. (2018). *Fault-Tolerant Message-Passing Distributed Systems.* Springer Nature Switzerland.

Spinellis, D., & Androutsellis, S. (2004). A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys, Vol 36, Issue 4* (pp. 335-371). ACM.

Tian, X.-M., Nemawarkar, S., Gao, G. R., & Hum, H. (1996). Data Locality Sensitivity of Multhithreaded Computations on a Distributed-Memory Multiprocessor. *Proceedings fo the 1996 conference of the Centre for Advanced Studies on Collaborative research* (pp. 37-50). Toronto: ACM.

Vitillo, R. (2021). *Understanding distributed systems.* Roberto Vitillo.