Section 4: Week 7: Tree-Structures and Fault-Tolerant Design

Nate Bachmeier

TIM-8120: Distributed Systems

November 3, 2019

North Central University

# Fault-Tolerant Design

## Influence of Hierarchy

Generally speaking, there are two mechanisms for modeling distributed systems, lists, and trees. A list can efficiently manage small groups of related nodes; however, it can become cumbersome with more massive sets. Trees allow for more expansive designs as the system can hierarchically describe the problem through multiple levels of control. Consider the difference between Domain Name Services (DNS, tree) and NetBIOS (list). NetBIOS can easily manage a small branch office, not the Internet, because its simple flat list structure is *globalized*. In contrast, DNS has multiple subdomains, with each subdomain owned by heterogeneous service providers. Since each subdomain holds a specific set of children, read and write operations can be *localized*.

## Influence of Partitioning

Localized designs are inherently more performant and fault-tolerant because of the containment of both scale and blast radius (Vosshall, 2018). Imagine a scientific dataset that has grown to several petabytes in size. The storage network would need to decompose this logical file system into multiple blocks and replicate it across multiple physical servers. These physical servers will run into mechanical failures, such as disk corruption or power outages.

When these outages occur, other nodes need to Setup, Challenge, and Repair (SCR) the missing data in an efficient manner (Chen & Curtmola, 2017). The time necessary to perform that repair operation is proportional to the size of each block and the system’s ability to scale the reconstruction over multiple peers horizontally. Assume that 1TB of the dataset has entered a failed state and needs to recover across a 10GB/s network (see Table 1). If only one virtual peer has a copy of the data, the system will heal in 102.4 seconds. Then contrast that with the smaller block size of 128GB and which can economically be sprawled across many servers, reaching an MTTR of under a second!

|  |  |  |  |
| --- | --- | --- | --- |
|  | Repair 1TB of Data | | |
| Block Size | **Virtual Peers** | **Num Blocks** | **MTTR (s)** |
| 1024 GB | 1 | 1024 | 102.4 |
| 8 | 1024 | 12.8 |
| 16 | 1024 | 6.4 |
| 512 | 2 | 2048 | 51.2 |
| 16 | 2048 | 6.4 |
| 32 | 2048 | 3.2 |
| 256 | 4 | 4096 | 25.6 |
| 32 | 4096 | 3.2 |
| 64 | 4096 | 1.6 |
| 128 | 8 | 8192 | 12.8 |
| 64 | 8192 | 1.6 |
| 128 | 8192 | 0.8 |

Table 1: Mean Time to Recover

## Influence of Fail-Over Groups

Proxy servers and similar brokers operate on ephemeral requests and need fault tolerance to come from a different source. One strategy is to maintain a target group of service instances and monitor their availability (see Figure 1). The monitoring can come from at least three reference points: (1) the network operating system, (2) the observed traffic of the broker itself, and (3) a local health agent on the service instance. The broker can use the Observed Health State Store (OHSS) to select the most appropriate receiver as new requests arrive.

A recovery policy could also exist to manage any Service Level Objectives (SLO) of the backend application. For instance, if the backend application needs to be highly available, the broker could be augmented to trap specific exceptions and automatically route to another node. Other systems need to optimize across different metrics, such as more consistent response time and choose completely different behaviors.

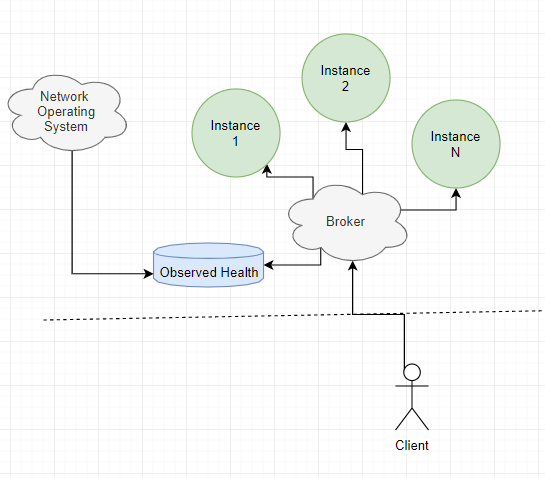


Figure 1: Broker Fail-Over

## Influence of Geo-Redundancy

Cloud Service Providers enable fault tolerance across multiple regions, so that entire data centers can fail without impacting applications uptime (see Figure 2). The scheme starts with deploying the service stack into two or more locations, such as Seattle and New York. Next, data store replication enables the sites to be kept in sync. Finally, the user can discover the most performant service stack instance from a location-aware Canolical Naming Service (CNAME). That system can consider latency and other metrics, similar to the proposed Fail-Over Group solution.

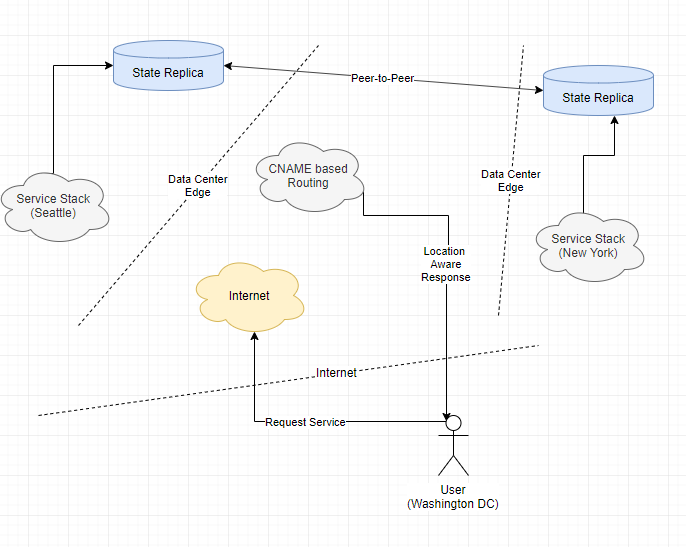


Figure 2: Multi-Region Deployment

## Influence of Consensus

The physical distance between the sites forces the need for eventual consistency protocols that range in complexity from (a) the latest timestamp wins, (b) Paxos algorithms, and (c) Byzantine General’s solutions (Zhao, 2014) The latest timestamp wins, is easy to understand but needs to rely on highly reliable distributed clock synchronization, an open research problem in itself (Ting, Chun-Yang, Di, Xiao-ming, & Heng, 2014) Under Paxos (see Figure 3), multiple rounds of preparation, acceptance, and learning phases take place to gain consensus. This elegant protocol can efficiently reconcile a single systems image, provided none of the nodes are malicious. If malicious or erroneous nodes exist, 3f +1 cross-validations need to occur (Zhao, 2014).

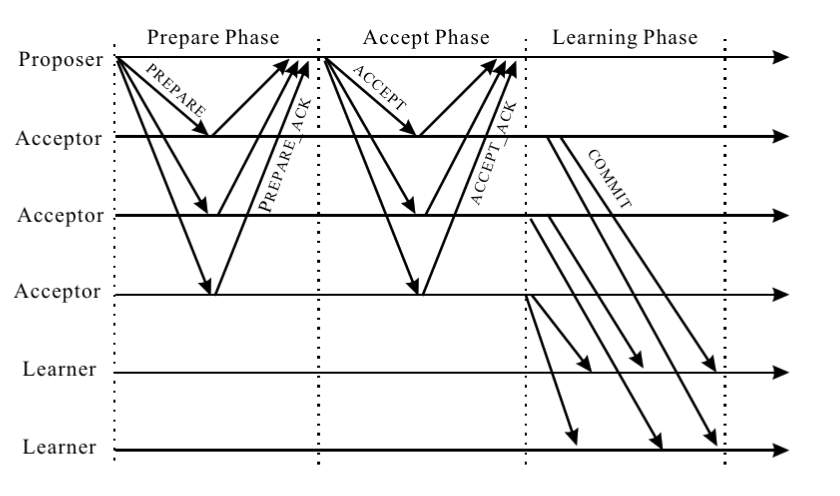


Figure 3: Paxos Consensus (Zhao, 2014, p. 196)

## Influence of Protocol

Message passing between components can either use reliable or unreliable communication. Unreliable handoff can be helpful for best-effort or performance-critical systems, such as real-time video or sampled telemetry reporting. Reliable handoff is crucial for scenarios that mandate full and consistent accounting, such as user data or financial records. These fault tolerance decisions are not limited to the low-level transport protocol differences between User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). They also appear at higher application levels (see Figure 4). The actor can notify the Alice service directly; however, the message could become lost due to a network failure. Instead, they can first place the payload into a command queue and remove it only after the server-side acknowledgment. When Alice accepts the event, it needs to receive confirmation from Bob and Charlie before returning success. Bob chooses to store the event in a durable command queue, versus Charlie executes it directly. In either scenario, the client can reliably infer that handoff has occurred.

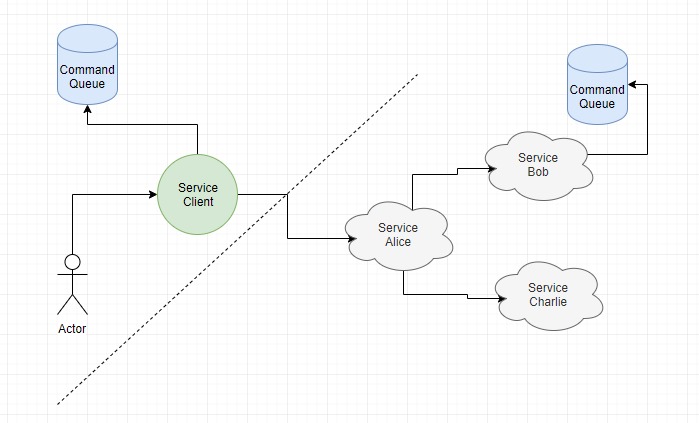


Figure 4: Handoff Protocol

# Data Structures for File Server Organization

A Distributed File System (DFS) needs to leverage multiple data structures to persist and retrieve content (see Table 2). The Contoso Network File System (see Figure 4) forces users to request a security token from their *Security Token Service (STS)*. The STS uses a B-tree to scan through an index to find the associated authorization policy. Next, the client can send their storage operation and user context to the *Storage Controller*. This operation places the payload in a Durable Priority Queue (DPQ) before acknowledging the client’s message received. A controller thread pulls the following DQP message and uses the *Storage Directory* to identify the detailed replicas. File objects can be arbitrarily large, with blocks spanning one or more modules. A tree structure maintains the replica information using linked lists at child levels for dynamic expansion across physical resources (see Figure 6).

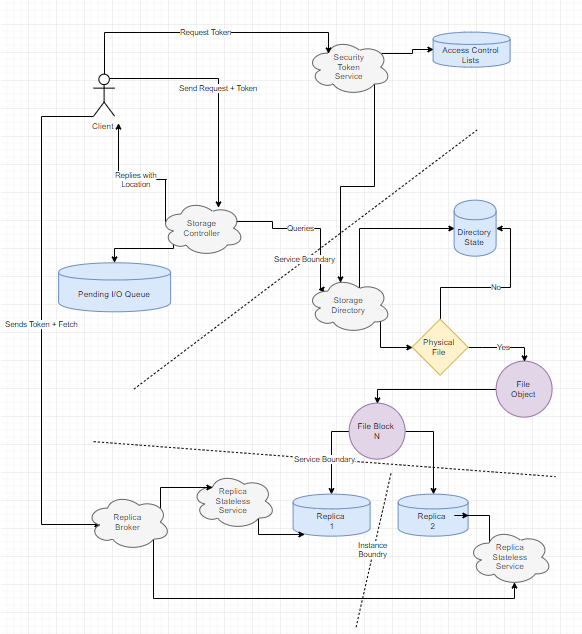


Figure 5: Contoso DFS Logical View

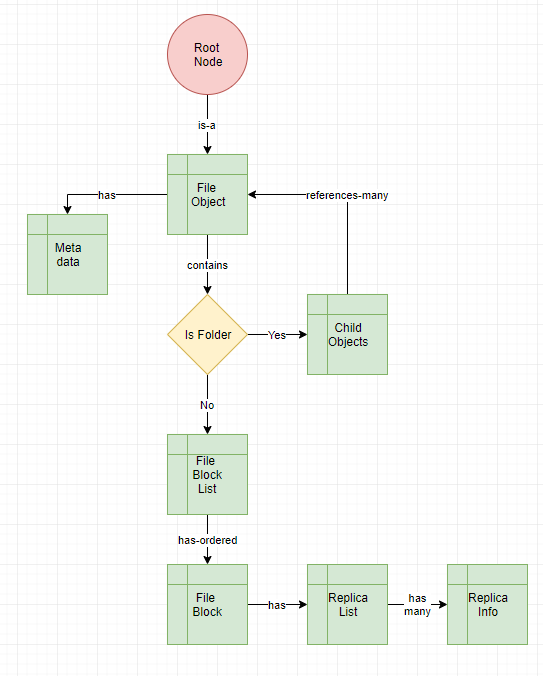


Figure 6: File System Tree Structure

|  |  |  |
| --- | --- | --- |
| Name | Description | Example Use Case |
| Trees | Parent-child relationship model | The hierarchical topology of object replica information |
| Stacks | Last-In/First-Out (LIFO) list | An ancestry list derived from a recursive traversal |
| Queues | First-In/First-Out (FIFO) list | An in-order pending operation buffer |
| Linked List | Flexible reorderable list | Ordered file list within a folder |

Table 2: File System Data Structures

Contoso’s DFS can support multiple concurrent application workloads, as there is a decoupling of incoming requests from event processing. This behavior is achieved through the Pending I/O queue that allows the service to remain highly available even during burst traffic. The system must maintain multiple copies of the content to ensure the high durability of a stored artifact. User traffic can be load-balanced across these different copies to increase read performance. Many scenarios, such as centralized service logs, can tolerate data being slightly out of order within the file. The DFS could support multiple concurrent writers to the same object by creating new blocks for each producer and recording them in the Storage Directory tree for these append-only use cases. These partial blocks reconcile Just-in-Time (JIT) for the consumer.

# Shortest Path Algorithms

The core objective of any shortest path algorithm is to find an efficient route across a network (or graph). It is essential to understand the applicability of these systems for numerous scenarios, such as routing packets or optimal multi-tenant utilization of resources. Dijkstra’s algorithm lives at the heart of many implementations (see Figure 5). Others have enhanced the solution with negative weights, priority queue filtration, and many-to-many pruning (see Table 3). These capabilities increase performance and decrease memory footprint, allowing the answers to scale to more extensive networks.

|  |  |  |
| --- | --- | --- |
| Name | Audience | Improvements Over Dijkstra |
| Dijkstra  (Computer Science, 2016) | - One-to-All Paths  - Positive edge weights only |  |
| Bellman-Ford (Sambol, 2015) | - One-to-All Paths | - Support for negative edge weights |
| A\* Search (Computerphile, 2017) | - One-to-One Path | - Introduces a heuristic to always move towards the goal  - Reduces memory and compute significantly |
| Floyd-Warshall (Sambol, 2016) | - All Pairs of vertices | - Builds a VxV matrix of all combinations |
| Johnson (Ravi, 2015) | - All Pairs of vertices | - Extends Floyd-Warshall to support negative weights - Applies transforms to make graph Dijkstra compatible |
| Viterbi (Chugg, 2017) | - Many-to-Many shortest path | - Uses a survivor pruning strategy to eliminate uninteresting paths - Supports state machine like transitions |

Table 3: Common Algorithms

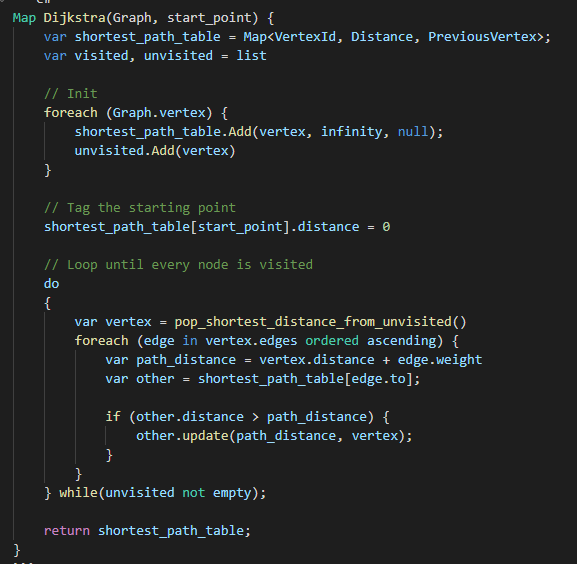


Figure 7: Dijkstra’s Algorithm

# Conclusions

Fault-tolerant design means different things for different use cases. Domain Naming Services (DNS) can rely on a hierarchical structure that scopes the ownership responsibilities within a heterogeneous collection of service providers. Distributed File Systems (DFS) need to be resilient to physical disk failure through replication. Files need to be broken down into small blocks and spread across many machines. This strategy helps reduce the MTTR and thus boosts the system's availability. Brokers need to introduce load balancing strategies across a pool of homogenous service instances. When one of these service instances fails, the broker can trap the exception and perform a custom policy to manage latency and availability trade-offs. Specific systems can also gain fault tolerance through geo-redundancy, so that entire data stores can go offline without impacting the service uptime. These strategies were limited to only large-scale enterprises but became affordable through Cloud Service Providers (CSP). Another critical approach for fault tolerance is handing consensus within the protocol, both at the transport and application layers.

Distributed File Systems need to use multiple data structures, of which trees are performance-critical and rationally organize data (see Figure 6). For instance, Contoso DFS uses branches for file blocks and another for the replica information of those blocks. When additional information needs to augment the structure, fine-grained locking can occur. Other data structures, such as priority queues and linked lists, are critical for building these systems and expressing the dynamic nature of the system. It would be challenging to implement these distributed systems without using shortest path algorithms. They appear in resource selection, traffic routing, state machine modeling, and lossy compression, among other scenarios. Dijkstra’s implemented the core algorithm, and it has been extended to support new behaviors and reduce search complexity.

# References

Chen, B., & Curtmola, R. (2017). Remote data integrity checking with the server-side repair. *Journal of Computer Security 25*, 537-584.

Chugg, K. (2017, March 8). *Viterbi Algorithm*. Retrieved from YouTube: https://www.youtube.com/watch?v=6JVqutwtzmo

Computer Science. (2016, May 7). *Graph Data Structure 4. Dijkstra’s Shortest Path Algorithm*. Retrieved from YouTube: https://www.youtube.com/watch?v=pVfj6mxhdMw

Computerphile. (2017, February 15). *A\* Search Algorithm - Computerphile*. Retrieved from YouTube: https://www.youtube.com/watch?v=ySN5Wnu88nE

Ravi, P. (2015, August 16). *Johnson’s Algorithm*. Retrieved from YouTube: https://www.youtube.com/watch?v=xc2ua8sQAoE

Sambol. (2016, July 16). *Floyd-Warshall algorithm in 4 minutes*. Retrieved from YouTube: https://www.youtube.com/watch?v=4OQeCuLYj-4

Sambol, M. (2015, June 3). *Bellman-Ford in 4 minutes - Theory*. Retrieved from YouTube: https://www.youtube.com/watch?v=9PHkk0UavIM

Ting, W., Chun-Yang, C., Di, G., Xiao-ming, T., & Heng, W. (2014). Clock Synchronization in Wireless Sensor Networks: A New Model and Analysis Approach Based on Networked Control Perspective. *Mathematical Problems in Engineering Volume 2014, Article ID 731980*, 1-19.

Vosshall, P. (2018, November 27). *How AWS Minimizes the Blast Radius of Failures*. Retrieved from YouTube: https://youtu.be/swQbA4zub20

Zhao, W. (2014). *Building Dependable Distributed Systems: Building Dependable Distributed Systems.* John Wiley & Sons, Incorporated.