Section 4: Week 8: Enhance a Distributed System Architecture

Nate Bachmeier

TIM-8120: Distributed Systems

November 10th, 2019

North Central University

# Enhance a Distributed System Architecture

Contoso Retail and Manufacturing is similar to many large enterprises in their need to manage and support heterogeneous networks that span multiple data centers and branch offices. Their workloads include standard services such as authentication, name resolution, and distributed storage system, along with proprietary systems. This topology naturally steers them towards distributed system designs, which come with their own sets of challenges, like balancing availability, reliability, and performance. Many of these challenges are NP-hard, meaning that the precise answers are complex to derive but easy to verify. Therefore they require efficient approximation algorithms or revolutionary changes in approach. Other aspects of the system need architectural patterns that enable Recovery-Oriented Programming (ROP)*.*

## What is NP Complexity

If the solution to a decision problem takes *polynomial time* to solve, then it would have a complexity of *P*. These questions would include addition, sorting a list, and performing many procedural transformations. Other problems take *nondeterministic polynomial time* to solve and fall into the category of *NP*. If the solution to an NP question is verifiable in polynomial time, then it is *NP-hard*. Examples include hardware verification, multiprocessor scheduling, and even Super Mario Brothers (Mann, 2017) (Aloupis, Demaine, & Guo, 2012). When both the answer and verification take nondeterministic polynomial time, these challenges are said to be *NP-complete*.

According to Aloupis et al., if and only if, a decision problem can be proven to be as tough as an existing NP-hard problem, can it be considered NP-hard. A common strategy is to map the new challenge to the classic *Satisfaction Problem* (SAT). SAT is known to be NP-hard as its “combination of Boolean variables, negations, disjunctions, and conjunctions […] cannot be solved more efficiently than exhaustive search (2^n steps) (Mann, 2017, p. 76).” Consider a Mario level (see Figure 1), where the player needs to collect (*True*) and use (*False*) power-ups in a precise sequence of events to reach the end. A model of these decisions can be completely mapped to the SAT and therefore said with certainty, to solve Mario is at least equally hard to solving SAT.

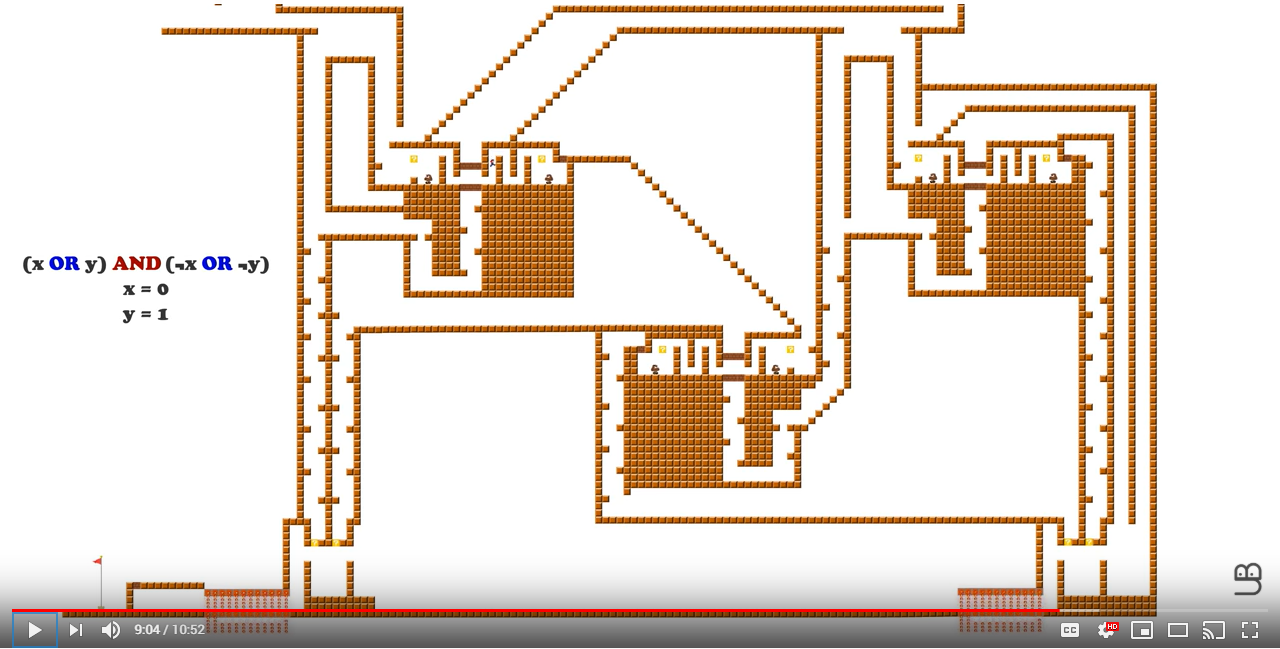


Figure 1 Super Mario as SAT (Undefined Behavior, 2019)

Mann states that there are many misconceptions around the notion of NP-hard and that literature frequently misrepresents the establishment of this assignment. Many scenarios, such as combinatorial enumerations, require exponential but finite time making them P. He also highlights that many problems *appear* similar, but can through changes to the model no longer be map precisely. These changes might include additional constraints to define the scope of the solution. Imagine the previous Mario level, where the player can walk through walls or use a ladder (oracle) to skip the maze and directly reach the end. Lastly, he cautions against the presumption that NP-hard can only solutions only exist through exhaustive search and heuristics. Many complex problems, such as bin packing, are addressable on a practical basis with a consistent selection strategy.

# What is Recovery-Oriented Programming

Fault-tolerant systems do not happen by accident and are the product of specific architectural patterns (Zhao, 2014). Zhao describes several variations of logging, checkpointing, isolation, replication strategies, along with implications such as group communication and consensus. A checkpoint must contain sufficient context for a recovery operation to continue the request at a later time. An isolation strategy will partition a logical resource into disjoined physical units so that the blast radius of a fault is manageable. Replication of both services and data provides a mechanism to absorb load and horizontally distribute it.

## Using Checkpointing

In Figure 2, an actor performs actions A-C and D-F with the fundamental difference being D-F creates state checkpoints between the operations. Consider the difference in recovery responsibility of a fail-stop between B-C and E-F. In the case of B-C, the error needs to be trapped and retried by the actor. Then contrast that with E-F where the responsibility lies with E. While it may be sufficient for specific scenarios to rely on the actor to make these recovery decisions, they become proportionally complex relative to the number of services involved.

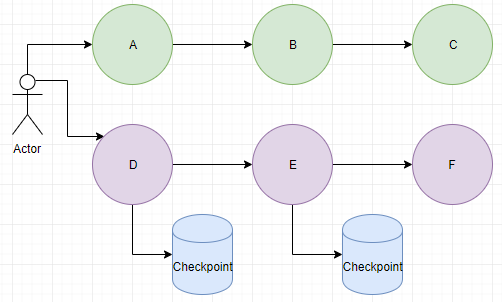


Figure 2: Checkpointing

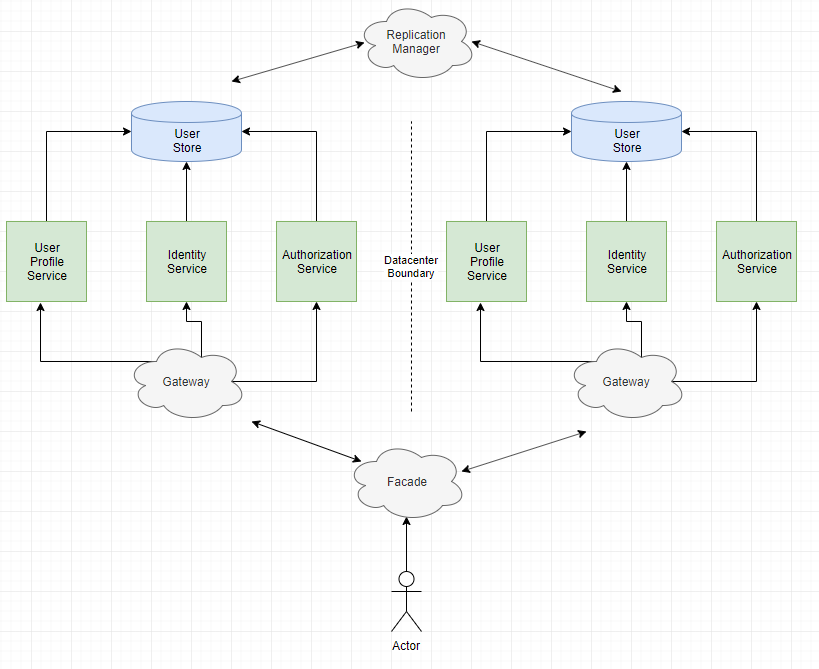


Figure 3: Isolation and Replication

## Using Service Partitioning and Replication

To ensure a collection of user services are highly available, the system designers can leverage service isolation and replication strategies (see Figure 3). This approach starts with decomposing the different domains of user management into separate micro-services, such as user profiles from authorization policies. Containerization and other virtualization technologies can create security barriers between each micro-service instance, such that the underlying physical hardware is sharable while minimizing additional risk (Qu, 2018) (Ahmad, Naveed, & Noda, 2018).

## Using Rejuvenation Policy

Load balancers and service orchestration engines can manage the lifecycle of these micro-services, through elastic provisioning and application of software rejuvenation policies (Yang, Min, Yang, & Li, 2013). For instance, a memory leak exists in the user profile service that eventually results in a fail-stop. The business cannot resolve the issue as they lack the expertise, due to the component using closed source software. Administrators mitigate the problem by provisioning multiple instances of the profile service, along with monitoring, and automation that recycles instances above X% committed memory. The micro-reboot of the service instance can then become managed and controlled. Instead of hard stopping, the automation can first take it out of the load balancer target group resulting in a natural drain of requests. Then after a specified duration or heuristic, the instance can be safely cycled.

## Using Datacenter Fail-over

Partitioning a service into multiple micro-services and then provisioning numerous copies of them, reduces the impact of an individual unit encountering a fail-stop scenario. While the strategy provides localized protections, they are ineffective against data center outages. Based on reporting from multiple news sources, it seems a significant disruption in a Public Cloud Service Provider (CSP) happens every six to twelve months (McGlaun, 2018) (Targett, 2019) (RIQ News Desk, 2016). Mitigation of these outages requires geo-replication of both services and data to multiple disjoined physical locations. For instance, the example user service could physically reside in Seattle and Boston. Then the actor can call into a service façade and rely on their traffic flowing to the closest availability zone (Qu, 2018).

## Challenges of Eventual Consistency

Some argue that only strongly consistent reads and writes to data stores should be used (Liu, Arden, George, & Myers, 2017). However, this assertion is challenging in geo-replicated data stores, given the latency and performance penalty of remote synchronization. Multiple protocols exist for partitioning the data within these stores and coordinating distributed transactions across them, each with different scenario optimizations (Bharati & Attar, 2018) (Zhao, 2014) (Almeida & Baquero, 2013). Almeida and Baquero propose a solution that relies on Eventually Consistent Distributed Counters (ECDC), where each node periodically recalibrates with their peers. Bharati and Attar expand on these ideas through a survey of open source technologies that use asynchronous Paxos, strict two-phase commit, optimistic concurrency, and even graph-based workload aware replication.

## Maintaining Consistency across Heterogeneous Datastores

Significant amounts of literature exist around implementing replication patterns across homogeneous datastores, yet Software as a Service (SaaS) paradigms are pushing the industry towards “built for purpose” strategies that span heterogeneous solutions (Brice & Idziorek, 2018). This evolutionary step also makes logical sense, as distributed systems rely on different data structures to efficiently perform specific expert scenarios. Consider the difference between a search and graph store, where one enables *term* *indexing* across a collection of documents, and the other explores entity *relationships*. The data within them might be from the same domain, but their access patterns require different technologies.

To address these replication challenges, researchers have looked towards centralized orchestration services (see Figure 4) (Limon et al., 2018) (Venkatesan & Sridhar, 2015). Limon et al. propose a multi-agent Saga solution for issuing actions and compensations. Venkatesan et al. describe a system built on Apache Orchestration Director Engine (ODE) and Business Process Execution Language (BPEL). Both systems track progress through a hierarchial in-flight transaction store with each service request starting a new branch. As the leaf nodes report success or failures, the status propagates to parent branches. If a branch enters into the failure state and a compensation action exists, it will be involved to retry or rollback the action operation.

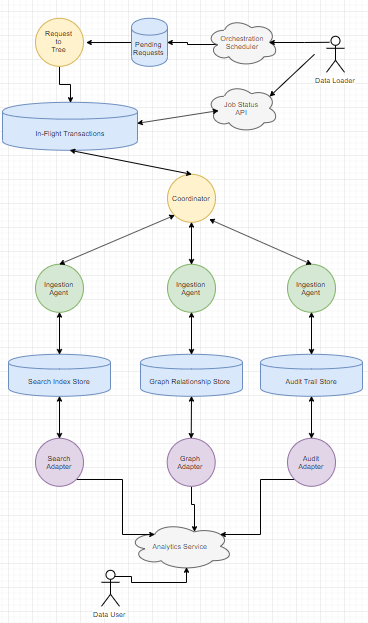


Figure 4: Multi-Agent Coordination

## Maintaining Consistency across Multi-Master Replicas

Contoso has a business requirement that its third-party product catalog is modifiable and queriable from a geo-aware replica. This need expands on the multi-agent solution to include Multi-Master Full Replication (MMFR)(see Figure 5). MMFR systems come with a unique set of challenges for maintaining reliability and performance. For instance, the need for more real-time updates increases the amount of communication and forces an asynchronous model (Macdeo et al., 2008). There are also additional complexities for traditional relational stores as each site must maintain enormous changelogs to rollback transactions (Filip, Vasar, & Robu, 2009). Amazon’s Dynamo service removes this constraint by broadcasting per item changes that using an array of vector clocks for versioning (Vogels, 2007). When a vector clock versioned item arrives, the local node attempts to prune competing instances and then allows the application to determine the final merge operation (e.g., last write wins or most recent not null value).

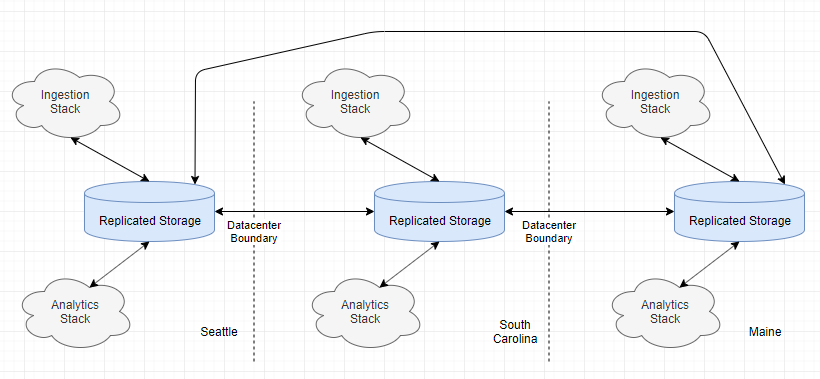


Figure 5: Multi-Master Full Replication Level 1

In Figure 6, the component level design shows the replication subsystem that exists between two geo-replicated environments. It relies on fully asynchronous communication by placing tasks inside of queues to signal that additional processing is needed (Alappatt, 2018). The first sink is the Pending Requests Queue (PRQ), which contains the requestor’s payload, version number, and security context. The Storage Router Service is a collection of elastic micro-services that scales proportionally to the queue depth (Sachs, Kounev, & Buchmann, 2013), and fetches any pending requests that are waiting for translation from the virtual object name into a physical storage node. The result of the storage operation, such as get data value or update a record, is given to the caller. When a storage node has acknowledged a change, it sends a replication request for that item into the Write Buffer Queue (WBQ). These are multicast to all peers that appear in the Peer Replica List (PRL).

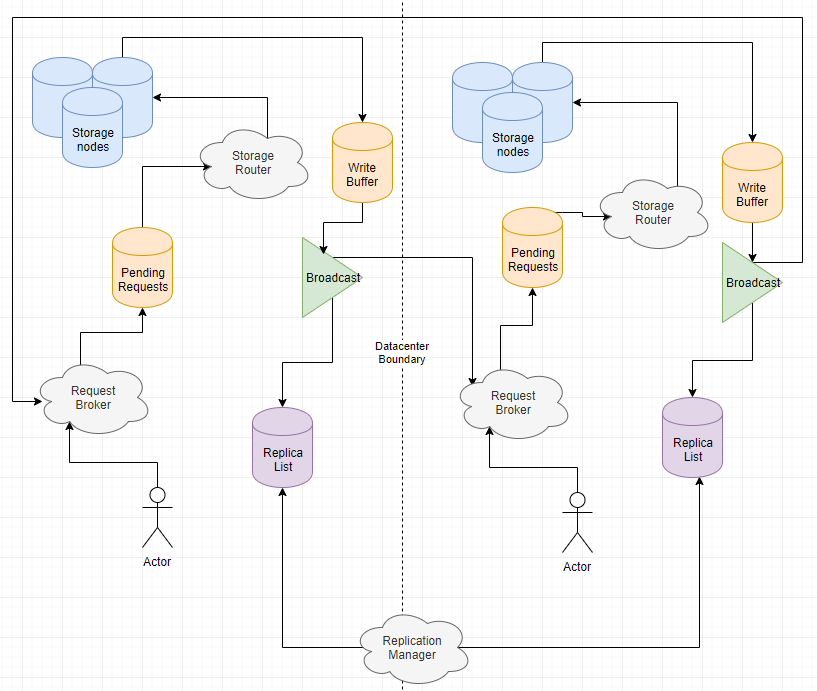


Figure 6: Multi-Master Full Replication Level 2