Distributed Systems

Christian J. Rudder

January 2025

Contents

Contents **1**

1 Introduction **5**

2 Working with Distributed Systems **6**

2.1 Replication: Synchronizing State 7

Bibliography **10**

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

— 2 —

Working with Distributed Systems

2.1 Replication: Synchronizing State

This section discusses replicating state across distributed systems.

Definition 1.1: Replication

Replication is the process of maintaining multiple copies of the same data on different nodes (machines). This is done for fault-tolerance, load balancing, and data locality.

Problem Space:
Consider we are running a money transfer service, from which Alice and Bob interact with:

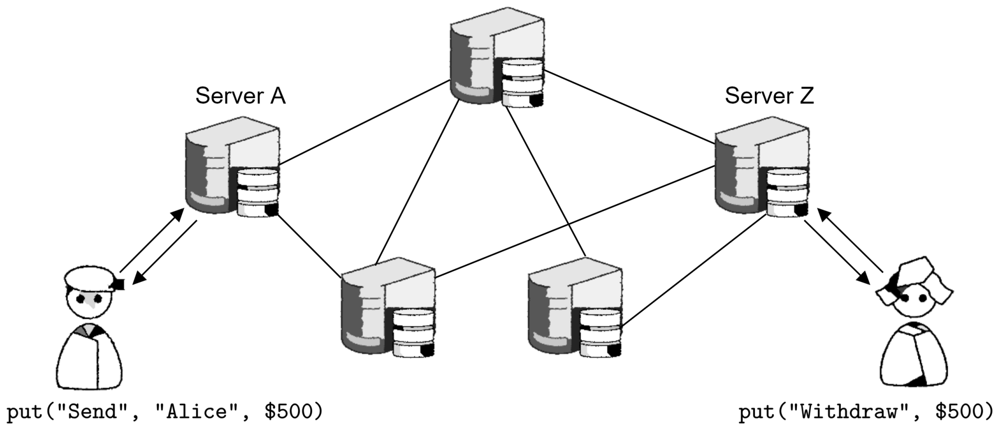

Figure 2.1: Bob sending money to Alice, while she withdraws money on two different servers.

In this scenario Many problems can arise: What if,

- The respective servers crash while or after Alice or Bob make requests?
- Alice and Bob share the same account, how do we ensure consistency?

In all, wow do we ensure the propagation and synchronization of state across multiple servers? We consider two models:

Definition 1.2: Active vs. Passive Replication

Active Replication: Client sends requests to all servers at the same time and waits for acknowledgments. This method must ensure that all requests are processed in the same order (expensive).

Passive Replication: Client sends requests to a primary server, which then forwards the request to backup servers.

We'll be move forward with **Passive Replication** as the preferred choice in this section. Though it is less expensive than active replication, it still has its challenges:

- **Consistency**: How do we ensure all backups are consistent?
- **Failure Handling**: What if the primary server fails?
- **Performance**: How do we ensure that the system is performant as we scale backups?

We consider two methods of replication:

Definition 1.3: State vs. Request Replication

State Replication: Forward the entire state to backups. This results in large message sizes, but is relatively simple depending on the system.

Request Replication: Forward only requests to backups. This results in smaller message sizes, but adds complexity when requests are not deterministic (e.g., random number generation).

Still again, we run into the following problem:

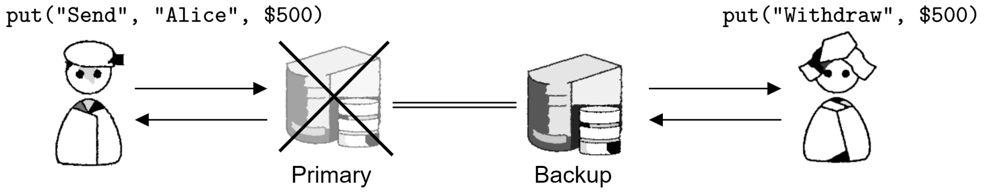

Figure 2.2: Bob's primary server failing, as Alice accesses the backup server.

How do we ensure both parties receive consistent feedback, even when the primary server fails?

Definition 1.4: Commit Point

The point at which the client is committed to a transaction, goes as follows:

1. The client sends a request to the primary server and waits for an acknowledgment.
2. The primary server forwards the request to the backup servers.
3. The backup servers process the request and send an acknowledgment to the primary server.
4. The primary server sends an acknowledgment to the client.

Step 4 is considered the **commit point**.

The following Figure illustrates the commit point in action:

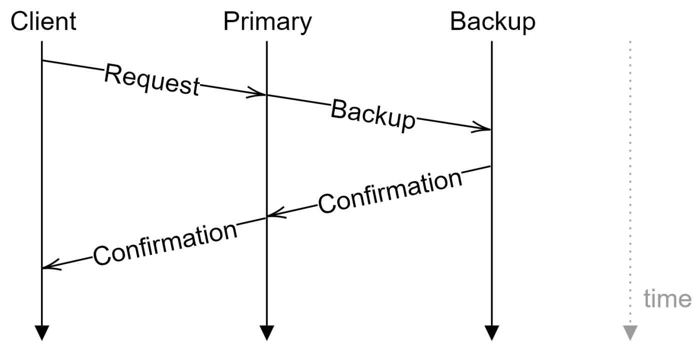

Figure 2.3: Client's requests propagating through the primary and backup sever.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.