Introduction to State Machine Replication

Learn about state machine replication—a general framework for building fault-tolerant distributed services.

We'll cover the following

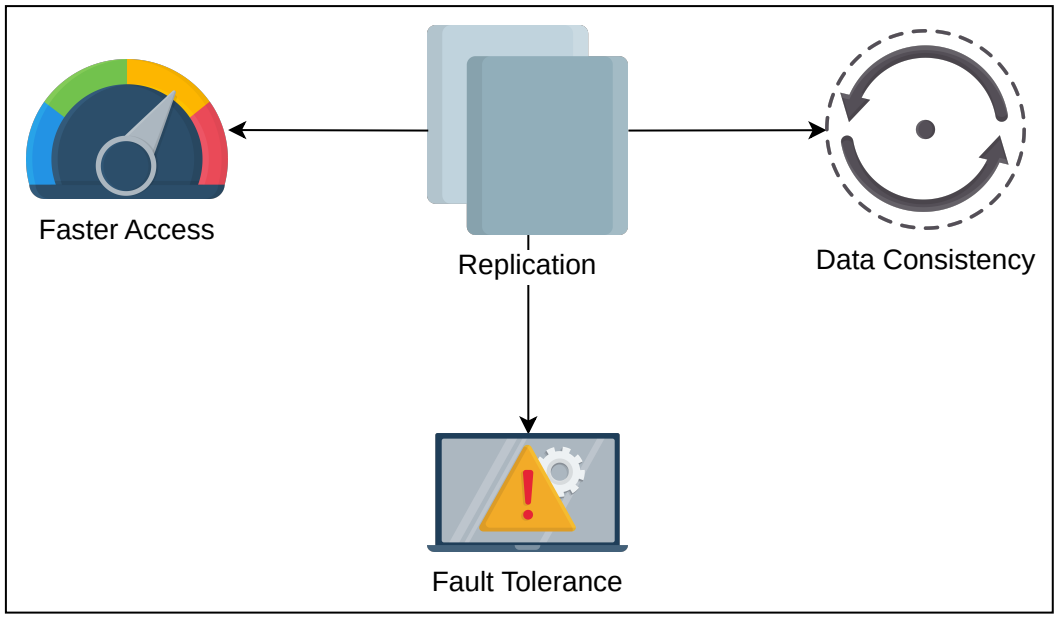
Motivation

Providing fault-tolerant services to the clients is a desirable property of a system. **State machine replication (SMR)** is a mechanism to implement fault-tolerant services. SMR models a system as a state machine and replicates multiple copies of these state machines such that failures are independent (meaning one failure only impacts one state machine). These state machines start with the same initial state, and the subsequent clients' requests reach every replica in the same order, which applies those commands to arrive at the same new state of the state machine (we are assuming deterministic logic that transitions the state machine from one state to the next).

The core component of SMR is the atomic broadcast facility, which enables every state machine to get the commands in the same order. NIST explains the purpose of state machine replication (SMR) as follows: “The objective of state machine replication (SMR) is to emulate a centralized service in a distributed, fault-tolerant fashion.” In this chapter, we will learn how to implement SMR when Byzantine and

fail-stop failures are possible and how to reconfigure a replica group by excluding the faulty nodes.

Replication is a widely used technique to design fault-tolerant distributed systems where we maintain replicas of data or services. Fault tolerance is a way to achieve higher availability and reliability. To tackle different kinds of failures, the replicated sites should be on different physical servers. Such servers might be in different data centers, which might be far apart.



While replication of a data service has many benefits, such as fault tolerance and faster read access, it also adds challenges to keep the replicated state consistent across all replicas and detect and replace faulty replicas with healthy ones

Example: Replicated database

Let's take the example of a database. A database has many data structures to efficiently facilitate clients' read and write requests. These data structures might constitute the state of the database. We can replicate this database at multiple places by replicating its state. When a client wants to read or write to the database, such a command is broadcast to all the replicas. Each replica, on receiving such requests in a specific order, executes it on the local state machine, and the state machine transitions from one state to the next. When all such requests have been applied, all the state machines should have the same state.

> **Note:** Remember that we assume deterministic transition logic so that all replicas could end up at the same state. By taking the majority vote of the outputs of the replicas, we could detect the one which had faults and didn't reach a correct next state.

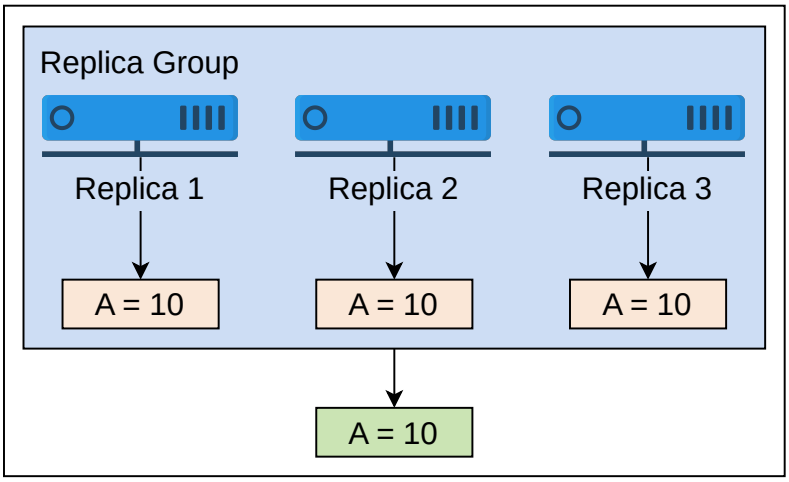
Before we define fault tolerance in detail, let's review different kinds of failure models.

Node failures

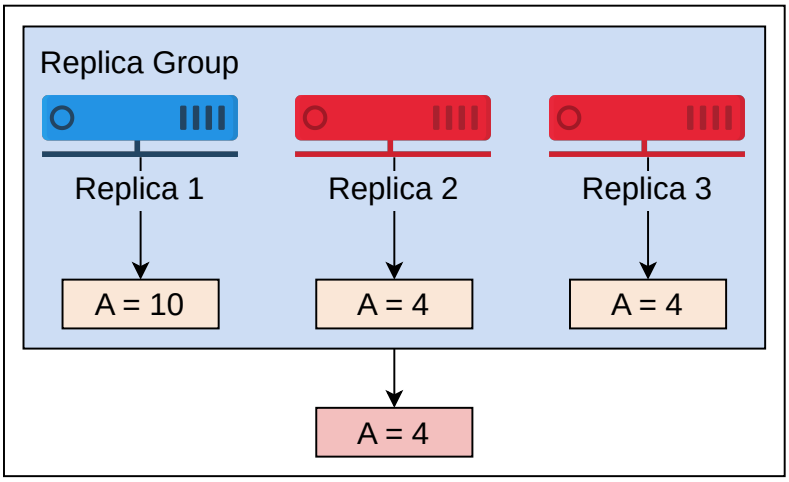
A component is considered faulty when its behavior is inconsistent with its specification. The entire spectrum of failures considered in distributed systems is covered by its two ends.

Byzantine failures

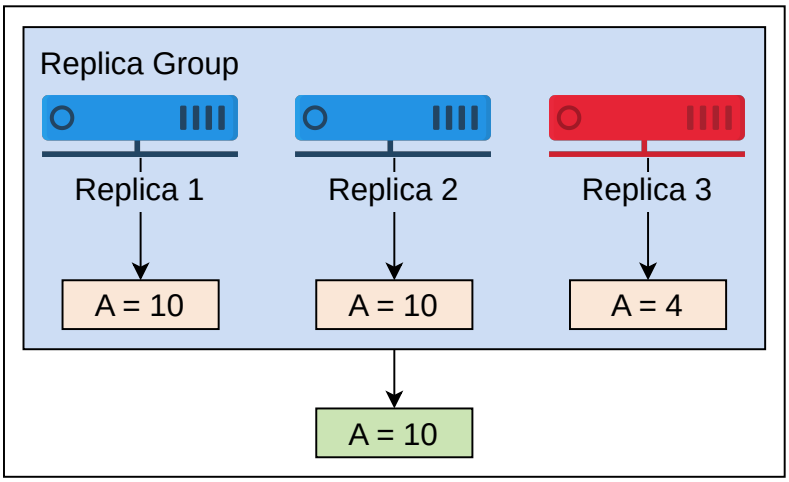
A **Byzantine failure** occurs in a node when it exhibits arbitrary behavior with other nodes, with the possibility of appearing to be working fine. It is not always possible to detect Byzantine nodes in a replica group.



In the above model, the value proposed by most replicas is considered the output of the replication group. Currently, all replicas are working fine and the output is correct.



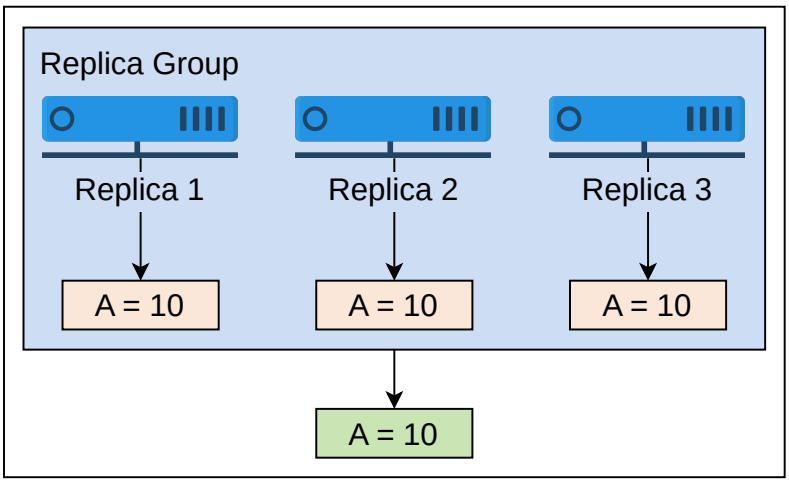
The replication group can give an incorrect output if enough Byzantine failures occur. We cannot differentiate faulty nodes from correct ones, so we will have to consider outputs from all nodes.



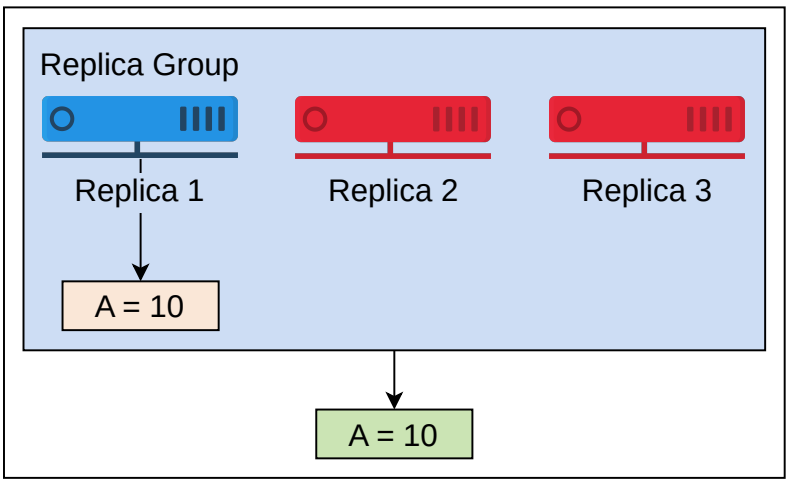
Later, we will see that one way to provide fault tolerance when Byzantine failures are possible is to ensure that the number of correct nodes is sufficient for the replication group to provide the correct value. In this example, with the total number of nodes being three, we need at least two nodes to ensure the system functions correctly.

Fail-stop failures

A **fail-stop failure** occurs in a node when its response to failure is to change to a state that reliably indicates its failure to other nodes.



Let's use the same system from the previous slide deck. The value proposed by a majority of replicas is considered the output of the replication group. Currently, all replicas are working fine and the output is correct.



When a node experiences fail-stop failure, it simply stops responding, and other nodes can determine its failure accurately. We simply use the output from the remaining replicas to provide the group's output. Later we will see that with fail-stop failures, we can provide fault tolerance by ensuring that there is always one correct node in the replication group.

Points to ponder

Question 2

Why didn’t we discuss possible network failures and only discuss node failures?

Hide Answer

The atomic broadcast subcomponent of SMR that broadcasts the clients’ commands in the same order to all the replicas in the group needs to handle all the possible network failures. Therefore, it is not true that SMR is not concerned about network failures.

Tolerating t failures

The motivation behind replication to improve fault tolerance is that a system with a single centralized server is as fault-tolerant as that server. For higher fault tolerance, we require replicas of servers that fail independently. This means that the failure of a replica should not depend on the failure of another. As a result, the system's failure is not contingent on one node failing but on many nodes failing. But how many nodes are allowed to fail before the system fails? We need a way to specify fault tolerance.

Traditionally, fault tolerance is specified as the **mean time between failures (MTBF)**. MTBF is an operational metric. Teams strive for maximum MTBF. While MTBF is a good way to measure fault tolerance, we can also specify fault tolerance in terms of the number of node or replica failures in a system, which comes with its advantages.

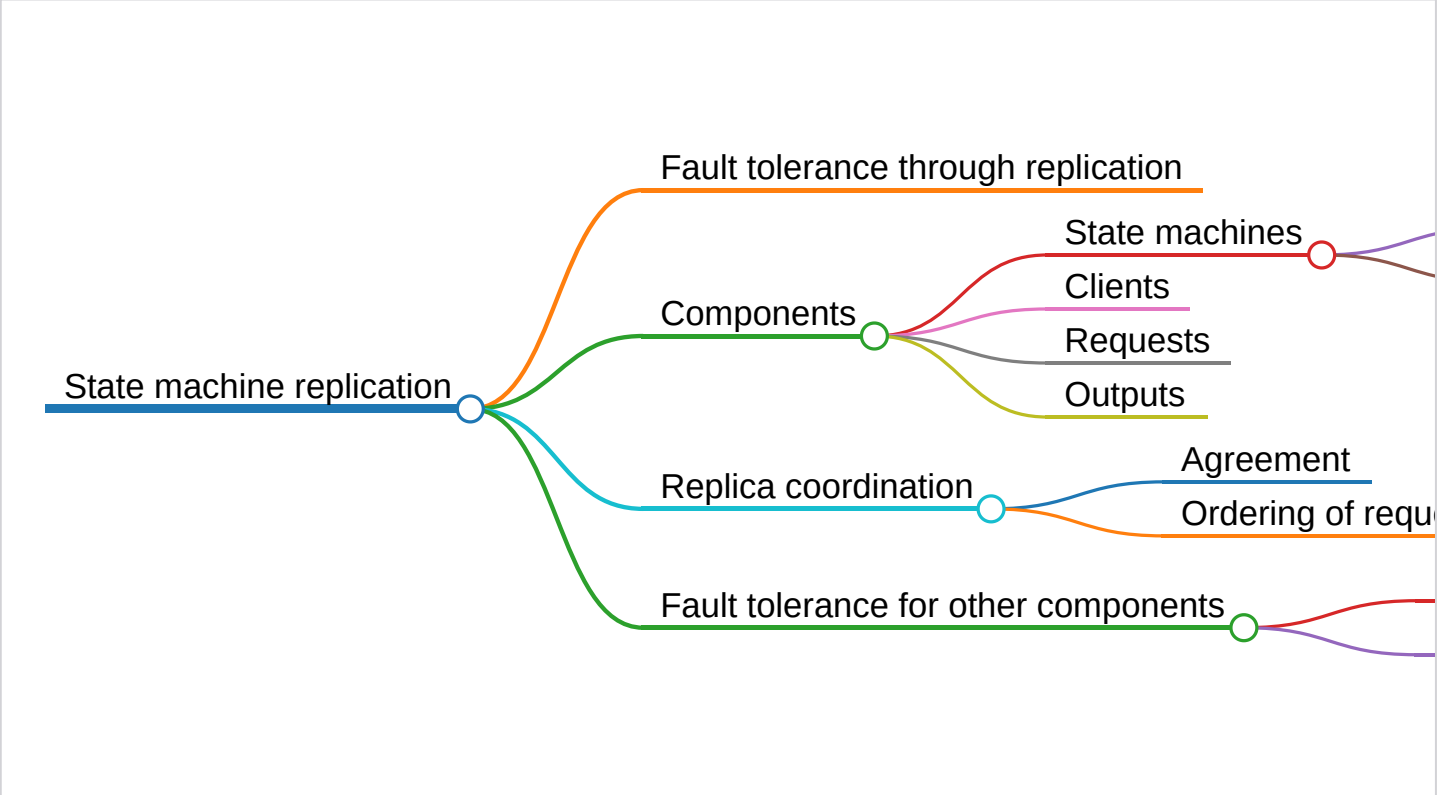
A system is considered to be t **fault-tolerant** if it successfully carries out its functions provided that no more than t nodes fail. A few advantages of specifying

fault tolerance using this approach are as follows:

1. This approach makes explicit assumptions about the reliability of a system which the MTBF approach does not. MTBF being average can be affected adversely by very high or very low values.
2. A system's t fault tolerance is unrelated to its components' reliability. This approach provides a measure of fault tolerance provided by the system's architecture rather than its components.

> **Note:** As a designer, knowing that a service can tolerate, for example, three independent node failures is more informative than saying that the MTBF of a service is two years.

Bird's eye view



An overview of what we will discuss in this chapter

The term state machine comes from finite state machines (or finite state automata). Our state machines in SMR can also have a finite amount of states like finite state

automata. Before delving deeper into SMR, we'll review some concepts regarding a state machine in the next lesson.

Back

Quiz on Two-Phase C...

Next

State Machines

Mark as Completed

Points to ponder

Question 1

Why are we just considering Byzantine and fail-stop models? What about the other models?

Hide Answer

Byzantine and fail-stop failures are two extreme ends of the failure spectrum. Byzantine faults are the hardest kind of failures to deal with, while fail-stop failures are easier to detect and manage.

In this chapter, we will see an SMR implementation with both failure models, which in some sense, covers the whole spectrum of failures. By understanding how to deal with the most difficult and the easiest types of failure models, we can effectively address anything that falls in between these two extremes.

Question 2

Why didn't we discuss possible network failures and only discuss node failures?

Hide Answer

The atomic broadcast subcomponent of SMR that broadcasts the clients' commands in the same order to all the replicas in the group needs to handle all the possible network failures. Therefore, it is not true that SMR is not concerned about network failures.