P2P Key-Value Storage System

Salvatore Gilles Cassarà,
Elia Gatti

I. Introduction

This report describes our implementation of the *Mars Distributed Storage System*, a "peer-to-peer key-value storage system". The system must be able to perform various operations on the nodes (Section III) and on the items in the storage (Section IV) while maintaining sequential consistency (Section II) and concurrent operations (Section IV-A). Moreover the storage system must also guarantee a certain degree of replication: given a replication factor *N*, each data item must be stored in *N* consecutive nodes.

A. Assumptions

Only the basic assumptions from the project description have been used:

- The network is assumed to be FIFO and reliable;
- There can not be less nodes than the replication factor *N*. Leave operations that would void this are aborted;
- There must always be at least one node in the distributed storage system that is not in crash state;
- Nodes can not crash during operations;
- The bootstrapping node received in a *Join* operation is a node currently active;
- To emulate network propagation delays, small random intervals are inserted between the unicast transmissions.

In order to take delays into account and distinguish them from crashed nodes that cause timeouts, we take advantage of knowing the maximum length of a delay $Delay_{max}$ and the number of messages exchanged for each operation Msg_{op} in order to define the timeout as:

$$Timeout_{op} = Delay_{max} \cdot (Msg_{op} + \epsilon)$$

This means that we consider that each message has maximum delay for all message exchange plus a small extra ϵ. If the operation takes more than this, we can be sure that the node has crashed.

B. Project Structure

The project is written in Java 21. The framework Akka 2.6 is used for the implementation and handling of the distributed system. The project is fragmented into various files, each one taking care of a different piece:

- *Main.java* sets the distributed storage and tests operations on it;
- *Messages.java* contains the classes of all the various messages, including their attributes and constructors;
- *utils/DataStoreManager.java* contains the replication factor and the read/write quorum;
- *utils/VersionedValue.java* defines the "VersionedValue" class, which is what we decided to use to store data in the distributed storage. It is a tuple (String, int) corresponding to (Value, Version);
- *utils/TimeoutDelay.java* contains the timeouts for the various operations computed as described in Section I-A and contains a method to Get the timeout time given the operation type;
- *utils/OperationDelays.java* contains the same timeouts but they also consider the messages exchanged with the client. It is used to be sure that the operation has been completed before printing the status and moving to the next operation;
- *types/OperationType.java* defines all the possible operation types;
- *actors/Client.java* has the Client class and the handlers for all messages received by the Client;
- *actors/StorageNode.java* contains the StorageNode class (a node belonging to the distributed storage) and the handlers for all messages received by the StorageNode.

A StorageNode can have 3 states (behaviours):

- *initialSpawn*: a node that has just been created but still hasn't joined the network or a node that left the network;
- *createReceive*: an online node connected to the distributed storage system;
- *crashed*: a crashed node which will ignore all messages until it recovers.

II. Sequential Consistency

With sequential consistency we mean that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [1]. In order to guarantee sequential consistency we associate a version to each data item in the storage and use quorum-based Get (read) and Update (write) operations.

The version of a data item allows us to recognize what is the most up-to-date item between its replicas in the distributed storage. When a Get or Update operation is performed, the coordinator asks for the requested data item to all *N* nodes containing its replicas. Once the coordinator has received at least *W* or *R* responses, the quorum has been satisfied and we can be sure that at least one of the responses contains the most

[1] How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

up-do-data data item, avoiding to work on older data items and as such not "traveling back in time".

In order to make the quorum work, the following constraints must be satisfied:

$$W > N \div 2$$

$$R + W > N$$

By setting both *R* and *W* to $(N \div 2) + 1$ we can be sure that both constraints are satisfied.

III. Operations on Nodes

A. Join

The *Join* operation consists in making a node join the distributed storage system. Due to the addition of a new node, data items need to be repartitioned: the new node takes data items that now belong to it and other nodes drop them to keep the replication factor at *N*. The message exchange happens as follows:

1) We start with a node in the *initalSpawn* state. We send it a *Join* message containing a bootstrapping peer;
2) The joining node contacts the bootstrapping peer and asks for its node registry (the list of nodes connected to the distributed storage system) though a *RequestNodeRegistry* message;
3) The bootstrapping peer answers with its node registry through an *UpdateNodeRegistry* message;
4) Once the joining node receives the list it saves it and adds itself to the list, then it sends a *RequestDataItems* message to what would be its clockwise neighbor based on its id, asking for its data items;
5) The clockwise neighbors sends back a *DataItemsResponse* message with its data items;
6) For each data items that the joining node would have to store, it performs a Get operation though a *ReplicaGet* on all *N* nodes holding it in order to be sure to find the latest version;
7) The *N* nodes send back their data item with a *GetResponse*;
8) The joining node checks the received data items, adds the most recent version to its data store and then sends an *Announce* message with its id to all nodes;
9) When a node receives the *Announce* message, all nodes add it to their node registry and if the new node has taken one of their data items and they don't need to store it anymore, they drop it.

B. Leave

The *Leave* operation consists in an active node leaving the distributed storage system. When this happens, the leaving node must give its data items to another node to maintain the replication factor of *N*. If there are *N* node in the system, the leave operation is aborted as it would be impossible to maintain the replication factor. The operation in details is as follows:

1) A *Leave* message is sent to the node we want to leave the network;
2) The leaving node creates a new node registry without it and checks whether the replication factor would still be satisfied. If it isn't, the operation is aborted. Otherwise, for each of its data items it takes the *N* nodes that would store them and sends them a *NotifyLeave* message to check whether they are active of crashed;
3) When a node receives a *NotifyLeave*, it simply answers with a *LeaveACK* message;
4) Once the leaving node has received all ACKs, it sends a *RepartitionData* message containing its data store to all nodes and leaves since we are sure that those node will not crash in the middle of the operation;
5) When a node receives a *RepartitionData* message, they remove the sender from their node registry and check if they would need to store some of the data items of the leaving node. If that is the case, they store them.

C. Crash

As briefly stated in Section I-B, we use Akka's *Receive* abstract class to define the state of a node. This class uses message matching builders to define the behaviour of a node. In particular, to make a node crash we send it a *Crash* message. When a node receives it, it simply changes its state through a *GetContext().become(crashed())*. In this new state, its behaviour is to ignore all messages apart from the *Recovery* one.

D. Recovery

During the recovery operation, a message is sent to the node that needs to be recovered. This message includes the id of the elected node that will assist in the recovery, known as the recoveryNode.

1) Recovery Process: Once the recovering node receives this message, it immediately changes its behavior to a $createReceive$ actor. This change allows it to handle all standard storage node messages. The first action it takes is to request the current status of the node registry from the recoveryNode. This registry is essential for determining which nodes are currently active. In response, the recoveryNode sends its own registry, which reflects the most up-to-date status.

2) Data Synchronization: Upon receiving the Updated registry, the recovering node performs a series of critical steps. It replaces its outdated registry with the one provided by the recoveryNode. To optimize performance, it then requests data items from its predecessor node in the ring topology, which is determined by the recovering node's ID. This is more efficient than querying every node that holds data. With the Updated topology, the recovering node can also safely delete any data items it is no longer responsible for. After the predecessor node has responded with its data, the recovering node adds only the data items it is now responsible for handling. Once its data store is corrected, the node is fully ready for use. No further changes to the node registry are needed, as the crash did not alter the state of the other storage nodes.

IV. Storage Operations

This section details the operations that constitute the Data Service interface. We will describe how data retrieval and storage are performed while ensuring consistency, even during concurrent operations.

A. Concurrent Operations

To handle concurrent Get and Update operations, which are central to the Distributed Storage System, we developed a two-level locking mechanism. This design prioritizes reducing message overhead at the expense of computational power. This trade-off is particularly justified in the proposed operating environment, where low-power devices make message transmission significantly more costly than local computation [2].

1) Coordinator-Level Lock: The first level is the Coordinator-Level Lock. This lock ensures that requests from multiple clients for the same key are handled correctly. It operates as follows:

- If an Update operation is in progress, the lock is exclusive, blocking all subsequent requests (both Get and Update) for the same key.
- If a Get operation is in progress, it only blocks incoming Update requests, allowing other Get requests to proceed concurrently.

Any request that encounters a lock and cannot proceed will fail after a timeout.

2) Replica-Level Lock: The second level is the Replica-Level Lock, which acts as the final consistency barrier. It enforces the same locking policies as the Coordinator-Level Lock. This second layer is necessary because a Coordinator may not be fully aware of other ongoing operations on the requested data, particularly if it does not currently handle the data itself. Despite the seemingly complex design, the logic is straightforward and allows concurrent operations without compromising data consistency.

B. Get

The Get operation consists of several distinct phases to retrieve data reliably and consistently.

1) Phase 1: Initiation and Coordination: The process begins when a Main node sends an *InitiateGet* message to the client, effectively starting a client-side timeout. Once the client receives this message, it sends a *ClientGet* request to the specified node, which becomes the coordinator for that request. The coordinator then generates an operation ID to track the request's status and checks for any ongoing Updates for the specified key. If none are found, it applies a coordinator-level lock for the Get operation. The coordinator then checks if it possesses the requested value and, if so, applies a replica-level lock. It then broadcasts a *ReplicaGet* message to all *N* nodes that are supposed to hold a replica of the data item and starts a timeout for the entire operation. If the coordinator is also a replica, it releases its replica-level lock after reading the value but maintains the coordinator-level lock.

2) Phase 2: Replica Response and Quorum: When a replica node receives a *ReplicaGet* request, it first attempts to apply a replica-level lock. If successful, it reads the value, releases the lock, and sends the value back to the coordinator in a *GetResponse* message. The coordinator adds each incoming *GetResponse* to the request's status. After each response, it checks if the read quorum has been reached. Any *GetResponse* message received after the quorum is met is simply ignored. Once the quorum is reached, the coordinator identifies the most recently Updated value among the received responses and sends it back to the client. Finally, it removes the stored operation status, thereby releasing the coordinator-level lock and completing the process.

C. Update

The Update operation is based on the read-before-write paradigm and is executed in distinct phases to ensure data consistency.

1) Phase 1: Initiation and Coordination: The process begins with an InitiateUpdate message at the main level, which starts a client-side timeout. Once the client receives this message, it sends a ClientUpdate request to the chosen node, which takes on the role of the coordinator. The coordinator generates an operation ID and checks for any ongoing operations on the key. If the key is free, it applies an exclusive coordinator-level lock for the Update. Because of the read-before-Update model, the coordinator tracks the read and Update portions of the operation separately, using the same operation for both.

2) Phase 2: The Read Phase: Following the lock, the coordinator performs a Get operation. This step differs from a standard Get in two key ways:

- When replicas read the value, they apply and hold a write lock.
- The coordinator requires a write quorum (W), rather than a read quorum, to proceed.

This read phase guarantees that the coordinator is working with the most recent version of the data.

3) Phase 3: The Write Phase: Once the quorum is met, the coordinator knows it has the latest value. It then writes the Updated value (or creates a new entry), sends it back to the client, and propagates the Update to all N replicas. As it sends the Update, it also releases the coordinator-level lock (also replica-level lock if used). When a replica receives the ReplicaUpdate message, it either Updates the value or creates a new entry, based on the provided key. After successfully writing the data, each replica also releases its replica-level lock, completing the process.

4) Timeout Handling: In a Update operation, several types of timeouts can occur. The first is a client timeout, which happens if the targeted node fails to respond, likely because it has crashed. The second type of timeout occurs when a request is blocked by a lock at either the coordinator or replica level. Since the process doesn't receive a response within the predetermined time window, the requesting node

[2]Prof. Picco's Introduction to Low-power wireless networking for the Internet of Things - Slide 32

sees it as unresponsive. This mechanism prevents a request from waiting indefinitely for a resource that is already locked by another operation. After the timeout is detected if there were some resources locked for example a coordinator-level lock encounters a already placed replica-level lock, the first one will be released.

D. Timeout Handling

Timeouts are a critical part of ensuring the reliability of the system, particularly during Get and Update operations. Two primary types of timeouts can occur:

1) Client-Side Timeout: This occurs when the client's initial request to a coordinator node fails to receive a response within a specified time window. This is a common indicator that the targeted node is likely in a crashed state.

2) Lock-Related Timeout: This happens when a request is unable to proceed because a lock is already in place at either the coordinator or replica level. Since the requesting node does not receive a timely response, it considers the operation unresponsive and triggers a timeout. This mechanism is crucial for preventing indefinite waits and deadlocks.

If a timeout is detected, any locks that were acquired by the timed-out operation are released. For example, if a coordinator-level lock was placed but the operation then stalled due to a replica-level lock, the coordinator lock will be released to prevent other requests from being blocked. Due to the exclusive nature of the locks used in the Update operation, it is more susceptible to these lock-related timeouts compared to the Get operation.

V. Notes on Logging

We used an external logging framework, SFL4J[3], to simplify debugging and better understand system behavior during development. We used four distinct logging levels to categorize messages by severity and purpose:

- Debug: This level provides verbose logging for detailed tracing of operations. It's best used with a low number of operations due to the high volume of messages it generates.
- Info: This level provides meaningful information about the system's state without the high message count of the Debug level.
- Warn: This level identifies non-critical errors or unexpected behavior that doesn't cause a program to crash but might need attention.
- Error: This level is reserved for explicitly defined error cases. These messages indicate when an operation has failed or when other serious issues have occurred.

[3]SFL4J's official website