Distributed Key-Value Store with Data Partitioning and Replication

Da Roit Stefano, Pasquali Thomas

Distributed Systems 1 - Project 2023

1 Project structure

The project is implemented in Java; we used **JavaFX** for providing a user interface suitable to show the state of the system in real time. The emulation of the distributed system is carried out through the **Akka** toolkit which provides an Actor-based programming model.

The structure of the project consists in four folders:

- **client**: contains the implementation of the UI for the clients.
- **main**: contains the implementation of the UI for the system. It is the entry point of the program.
- **system**: contains all the logic implementation of the system, independent from the UI.
  - *KeyValStoreSystem.java* is useful for the management of the system since it is responsible for the creation of nodes and clients, together with the communication between them.
  - *Node.java* provides the implementation of the nodes in the distributed system, including the messages they can send to each other.
- **tests**: provides the implementation of some tests on the system; in particular related to sequential consistency and crash-recovery process.

2 System design

The system involves two types of entities: **Node** and **Client**. Clients can create **Get** and **Update** requests by sending to a Node a message which becomes the coordinator of the request. To support all the functionalities required by the project the following message types have been defined.

2.1 Public interface (Client/Main to Node)

- **Get**: sent by Client to a Node (coordinator); the message contains a key of the item requested.
- **Update**: sent by Client to a Node (coordinator); the message contains a key value pair for the write operation.
- **Feedback**: sent by Node to client; the message contains a feedack for the requested operation. In case of read operation, the message contains the value of the item requested; for the write operation there is only the result.
- **NodeLeave**: sent by Main to a node to inform it has to leave the system.
- **Crash**: sent by Main to a node in the system to inform it has to crash.
- **Recovery**: sent by Main to a node in crash status to trigger the recovery process. Includes the reference to a node in the system to contact for the current list of nodes.

2.2 Private interface (Node to Node)

- **NodeHello**: sent by a joining node to all the others in the system to announce itself as a new member. Includes the ID of the new node.
- **NodeGoodbye**: sent by a leaving node to all the others in the system to announce that it is leaving. Includes the ID of the leaving node and the items (key-value pairs) to store in the receiver nodes (the receivers become responsible of items stored by the leaving node).

- **GetNodes**: sent by a joining node to a bootstrap node to request the list of nodes in the system.
- **GetNodesResponse**: response of the bootstrap node containing the ID of the bootstrap node and the list of nodes.
- **GetItems**: sent by a joining node to the nodes responsible of its items to request them.
- **GetItemsResponse**: response of a *GetItems* containing the items to store in the new joining node.
- **GetItem**: identified by a request ID, it contains an item's key and the act requested (get/update); it is sent by the coordinator node to the others to query the value of an item.
- **GetItemResponse**: identified by a request ID, it contains an item's value; it is sent by the nodes which receive a *GetItem* to the coordinator node including the value of the item requested.
- **UpdateItem**: sent by the coordinator node to the others to update a value in the store.

2.3 System operations

2.3.1 Node Joining

1. The main creates a new node which during its pre-start operations contacts a set bootstrap node to request the list of the current nodes which compose the system by a *GetNodes* message. The bootstrap node will answer sending a *GetNodesResponse* message.
2. After the reception of the list of nodes, the joining node selects the replicas which should hold its interested items and sends them a *GetItems* message to obtain all the updated versions. The contacted nodes will select the items in the store that should be handled by the joining node and will reply attaching them to a *GetItemsResponse* message.
3. At this point the joining node waits for the reaching of the reads quorum. Whenever a new message arrives, it updates its own store based on the received items. When the quorum is reached (or after a timeout) the new node announces itself to the whole system by a *NodeHello* broadcast message containing the identifier of the joining node.
4. When the other nodes receive the *NodeHello*, they select the items which are no longer responsible for and remove them from the store.

2.3.2 Node Leaving

1. The main sends a *NodeLeave* message to the node which has to leave the system.
2. The node scans all its items and selects the nodes which will inherit their responsibility. After that, the node announces to every other node that is leaving by a *NodeGoodbye* message, attaching its items (only if needed) to any node that becomes responsible for them.
3. Upon the reception a *NodeGoodbye* the nodes put the items attached to the message in their store and remove the leaving node from the list of nodes.

2.3.3 Read Item

1. A client sends a *Get* message to a selected coordinator node with the requested item's key.
2. The coordinator node creates a *GetItem* message including a request ID (unique for the coordinator) and the key requested by the client. At the same time the coordinator instantiates a *PendingRequest* of type GET identified by the just mentioned ID and selects the responsible nodes of the item requested. After that, it sends the created message in multicast to them and starts a timer that will fire after T milliseconds.
3. The nodes which receive a *GetItem* message, read the item requested from the store, and include it in a *GetItemResponse* message.
4. The coordinator waits for *GetItemResponse* messages; upon a successfull *GetItemResponse* (identified by the unique request ID), the value for the quorum is incremented and the response value is registered. When the quorum threshold is crossed, the coordinator selects the most recent value from the list of received items comparing their version attribute. At the end, a *Feedback* message is sent to the client and the *PendingRequest* is deleted.
5. If the timer fires before reaching the quorum, an error *Feedback* message is sent to the client and the *PendingRequest* is deleted.

2.3.4 Write Item

1. A client sends an *Update* message to a selected coordinator node with the item's key to update and its new value.
2. The coordinator node creates a *GetItem* message including a request ID and the key requested by the client. At the same time the coordinator instantiates a *PendingRequest* of type UPDATE identified by the ID; then, it selects the responsible nodes of the item to update and sends the created message in multicast to them. The coordinator starts a timer that will fire after T milliseconds.
3. The nodes which receive a *GetItem* message read the item requested from the store and include it in a *GetItemResponse* message.
4. The coordinator waits for *GetItemResponse* messages; upon a successfull *GetItemResponse* (identified by the request ID), the value for the quorum is incremented and the response value is registered. When the quorum threshold is crossed, the coordinator sends a *Feedback* message to confirm the update operation. Furthermore, it increments the version of the item after selecting the latest one from the list of received items.
5. Then, the coordinator carries out the update multicasting to the responsible nodes an *UpdateItem* message including the new value and the computed latest version.
6. The nodes which receive an *UpdateItem* message save the item in the message in the store.
7. If the timer fires before reaching the quorum, an error *Feedback* message is sent to the client and the *PendingRequest* is deleted.

2.3.5 Crash and Recovery

1. A node can receive a *Crash* message which causes the receiver to pass in the crash status. During the crash status a node can handle only *Recovery* messages, while the others are ignored.
2. The main can send a *Recovery* message to a node in crash status to trigger the recovery process; the message includes the reference to a node in the system to whom send a *GetNodes* request to obtain the list of current nodes.
3. Upon discovering which nodes are in the system, the recovering node removes from the store the items it is no longer responsible for and requests the items it became responsible for by a multicast of *GetItems* message.
4. For each *GetItemsResponse* message it receives, the recovering node updates the local store. The recovering process ends when the quorum of reads is reached.

2.4 Simulation and tests

The UI of the system consists in a window which shows all the nodes involved. Every node has two sections: one shows all its logs printed during the execution, useful to track the ongoing operations, and one provides the items in its store. To interact with the system, we can use directly the buttons provided by the UI which allows to perform all the tasks supported by the system. Every client consists in a single window where we can trigger the read and write requests to the nodes; the results are showed in the logs section of the window.

The abstract class *Test.java* is used to implement automated tasks to test the system. In particular, we provide a test for sequential consistency which cannot be replicated using the UI, since the requests have to be concurrent. The test consists in multiple concurrent writes and reads/writes exploiting also delays. The CrashRecovery test provides the execution of a scenario in which a node crashes multiple times while other nodes join and leave and clients perform updates. During every test the state of the system can be changed after every step using the UI, since the progress of the execution has to be confirmed by the user.

3 Implementation

3.1 On Sequential Consistency

3.1.1 Problematic scenarios

- Write/write conflicts $\longrightarrow$ when two or more write operations on the same item are executed concurrently, there is no assurance that every replica ends with the same value, since the order of reception of the messages may be different for every node. Potentially we could have different values for the same version of the item.

- Write/read conflicts $\longrightarrow$ the sequential consistency can be broken when write and read operations are executed concurrently on the same item. The new value may be read by a client while some nodes have yet to receive the update; if the client reads the new value and then reads again, it is not assured that the old value is not the one returned. This may occur in a very specific scenario. Let's assume that:
  - the coordinator of the update is one of the node responsible for the item involved
  - the coordinator of the update has already upgraded its value and has just sent the *UpdateItem* messages
  - the other replicas have not yet received *UpdateItem* from the coordinator

  If two read operations of the same client on the same item overlap, $R < N$, and the coordinator of the concurrent update is not involved in both the read operations the following scenario could happen:
  - the first read can return the new value (coordinator of the update involved in the quorum)
  - the second one could read the old value (coordinator of the update NOT involved in the quorum)

3.1.2 Solution

To assure sequential consistency in the system, we provide for every node a *pendingUpdates* set of integers, useful to lock an item during write operations. More precisely, when a node

- receives a *Update* message from a client (the coordinator node)
- receives a *GetItem* message related to an update operation from a coordinator

the item's key involved in the write operation is put in *pendingUpdates*. The key is not removed from the set until the write is committed by the coordinator with an *UpdateItem* message or the timeout occurs. If a concurrent write/read on the same item is attempted, at reception of a *GetItem* message every node will check its set of locked items; if the key is locked, the node will not answer to the coordinator. In this way, we assure that at most one single write operation is carried out on the same item and that concurrent reads during an update cannot occur on the same key.

3.2 Analysis on system messages

During read and write operations, the number of messages exchanged is minimal since the coordinator sends the messages only to the responsible nodes for the item involved; if the operation is on a local value of the coordinator the quorum is immediately incremented saving up the sending of a message to itself.

Same happens for join and leave operations. The joining node requests the items only to the ones which are responsible for its interested values. The items are sent to the joining node all together in the same message (not a message for each item). The same holds for the leaving operation: the leaving node sends its items only to the new responsible nodes for them through a *NodeGoodbye* message. If the receiver node does not inherit items from the leaving node the message is empty.

3.3 Additional assumptions

To the assumptions listed in the project's description document, we added the following:

- The replication parameters can be set at compile time, but the following conditions must hold to assure a working system: $R + W > N$ (to avoid read/write conflicts) and $W > \frac{N}{2}$ (to avoid write/write conflicts). These constraints guarantee that at least one replica in the quorum contains the most recent write.
- An *UpdateItem* message can be received at most after T ms from the sending of a *GetItemResponse* message. This is necessary to guarantee sequential consistency with concurrent requests on the same item since after a timeout of T ms the interested item is unlocked anyway to avoid deadlocks.
- A crashed node is not really crashed, but discards all messages that are different to the *Recovery* type.
- If the key of an item is greater than the maximum value of the nodes' identifiers, the item is assigned to the node with the lower identifier.