Distributed Key-Value Store with Data Partitioning and Replication

Giacomello Anna - Vecellio Reane Martina

Distributed Systems I, 2022-2023 - Project

1 Structure

The Java programming language was used to create the "Distributed Key-Value Store with Data Partitioning and Replication" project. Message exchange and concurrency was implemented using the Akka toolkit, in order to have an Actor-based project.

The project structure consists in the following classes:

- **Client** is used to manage and respond to messages of two specific types: messages from ring to main and error messages.
- **DHTSystem** is the main from which all requests start, where the graphical interface is created.
- **Request classes**:
  - **Request** for read and update requests.
  - **ExternalReqeust** that includes:
    - **JoinRequest** for the join request.
    - **RecoveryRequest** for the recovery request.
- **Item** is used to maintain the storage of the nodes (i.e., key, value and version) and manage the lock/unlock of the items during the various operations to guarantee sequential consistency.
- **Peer** is used to associate each actor with their own ID.
- **Ring** includes the entire logical implementation of nodes, messages and their respective operations.
- **Test** is used to test system operations and messages, in particular to test sequential consistency and crash/recovery operations.

2 Messages Used in the System

2.1 External Messages

External messages are messages that are exchanged between nodes and main or between nodes and clients.

- **StartMessage**: start message that sends the list of participants to everyone and sets the initial storage.
- **GetValueMsg**: message to send a read request specifying the key of the value to read.
- **UpdateValueMsg**: message to send an update request specifying the key and the value to update.
- **ValueResponseMsg**: message to send the response for read and update operations to the client if the quorum is reached.
- **JoinRequestMsg**: message used to send a join request.
- **LeaveRequestMsg**: message used to send an announce to leave.
- **CrashRequestMsg**: message to send a crash request.
- **RecoveryRequestMsg**: message to send a recovery request.

2.2 Internal Messages

Internal messages are messages that come between two nodes.

- **GetValueMsg**: message used to send read requests during recovery and join operations.
- **RequestAccessMsg**: message used to determine whether access is granted or denied based on the type of request and the availability of the requested item, and to communicate the access status to the coordinator actor.
- **AccessResponseMsg**: message to send request to nodes that have the requested item to find the latest version if access is granted; otherwise, the request is added to the queue.
- **RequestValueMsg**: message to create a new item if the key is new, and to send a response containing the requested value to the sender.
- **ChangeValueMsg**: message used to update or create the item object of the update operation.
- **TimeoutRequest**: message used to schedule a timeout after a specified time for read and update requests.
- **TimeoutExternalRequest**: message used to schedule a timeout after a specified time for external requests (i.e., join, leave and recovery).
- **TimeoutDequeue**: message used to handle the timeout event for dequeuing requests.
- **UnlockMsg**: message used to unlock an item.
- **OkMsg**: message used tocount the number of resposes for an update request, used to remove the request and unlock the item when quorum is reached.
- **SendPeerListMsg**: message used to find the clockwise neighbor and to send a message in order to retrieve the list of peers.
- **GetNeighborItemsMsg**: message to send the list of items.
- **SendItemsListMsg**: message used to insert the list of items in the storage and to send a read request for each item.
- **AnnounceJoiningNodeMsg**: message used to announce a new joining node.
- **ReturnValueMsg**: message used to update the local storage.
- **AddItemToStorageMsg**: message used to add an item to the storage.
- **AnnounceLeavingNodeMsg**: message used to announce a leaving node.
- **GetPeerListMsg**: message used to get the peer list.
- **SendPeerListRecoveryMsg**: message used to send the peer list.
- **GetItemsListMsg**: message used to get the list of items.
- **SendItemsListRecoveryMsg**: message used to send the list of items.

3 Operations

3.1 Read Operation

1. The client sends a *GetValueMsg* message to the coordinator with the key of the value to read.
2. The coordinator creates a new request that includes the key of the value to read, and a timeout is set which will trigger after 5000 ms. Then, the request is sent to the owner of the item (i.e., the first node that has the item) with a *RequestAccessMsg* message.

3. The node that receives the message (i.e. the owner) checks that the item with that key exists, otherwise sends an *AccessResponseMsg* indicating that the access was denied and including a boolean flag that indicates that the item was not found. Otherwise, it checks whether it can access the requested object via a locking mechanism (i.e., if there is no update in progress on that key). Finally, an *AccessResponseMsg* is sent to the coordinator, also containing *accessGranted*, a Boolean value indicating the result of the lock operation (i.e. whether the access to the item was granted or not), and a Boolean flag indicating that the item was found; this flag, already mentioned in the sentence above, is useful to discern the situations where *accessGranted* is false because the item was already locked in a way that was incompatible with the requests, and the situations where *accessGranted* is false because the item was not found.

4. Once the coordinator receives the message, if access is granted, it sends *RequestValueMsg* to each of the nodes that owns this item; otherwise, it removes the message from the active requests and places it in the request queue.

5. Each node that receives the *RequestValueMsg* retrieves the item with the specified key and sends it back to the coordinator via the *ValueResponseMsg*.

6. Once the coordinator receives the *ValueResponseMsg*, it takes care of keeping the item with the most recent version and increasing the number of responses received. When the number of responses received is greater than or equal to the quorum, the request is removed from the active requests and the item is unlocked, and a *ReturnValueMsg* message is returned to the client with the most updated item corresponding to the requested key. If the quorum is not reached, the timeout will occur after 5000 milliseconds.

3.2 Write Operation

1. The client sends a *UpdateValueMsg* message to the coordinator with the key of the value to update or add and the value to insert.

2. The coordinator creates a new request that includes the key of the value to update and the value to insert, and a timeout is set which will trigger after 5000 ms. Then, the request is sent to the owner of the item (i.e., the first node that has the item) with a *RequestAccessMsg* message.

3. The node that receives the message (i.e. the owner) retrieves the item from the storage. If the key does not exist in the storage, the owner creates a new item with that key and version 0. Then, it checks whether it can access the requested object via a locking mechanism (i.e., if there is no reads and updates in progress on that key). Finally, an *AccessResponseMsg* is sent to the coordinator, also containing a Boolean value indicating whether access to the item is granted or not.

4. Once the coordinator receives the message, if the access is granted, it sends *RequestValueMsg* to each of the nodes that owns this item; otherwise, it removes the message from the active requests and places it in the request queue.

5. Each node that receives the *RequestValueMsg* retrieves the item with the specified key and sends it back to the coordinator via the *ValueResponseMsg*. If the item does not exist, a new one is created with version 0.

6. Once the coordinator receives the *ValueResponseMsg*, it takes care of keeping the item with the most recent version and increasing the number of responses received. When the number of responses received is greater than or equal to the quorum, a message to the client is sent with the most updated value and version of the item.

7. The request is removed from the active requests and added to the pending requests, and a *ChangeValueMsg* message is sent to each node that owns the item with the value to be changed and the most recent version. If the quorum is not reached, the timeout will occur after 5000 milliseconds.

8. Each node that receives a *ChangeValueMsg* updates its item and version and sends on *OkMsg* to the coordinator.

9. When the coordinator receives $\frac{N}{2} + 1$ *OkMsg*, it unlocks the item and removes the request from the pending requests.

3.3 Join Operation

1. The client sends a *JoinRequestMsg* message to the bootstrapping peer with the new peer.
2. The bootstrapping peer receives the request and checks that the new peer's ID is not already in use by another node. If the ID is unique, the request is created and a *SendPeerListMsg* is sent to the joining node, containing the list of peers.
3. When the joining node receives the peer list, it adds itself to the peer list and updates it in order. It then finds the clockwise neighbor node and sends it a *GetNeighborItemsMsg* to get the list of items. A timeout is also set, which kicks in after 5000 ms.
4. When the clockwise neighbor node receives the message, it sends the joining node the list of items it possesses via *SendItemsListMsg*.
5. Once the joining node receives the list of items, it sets its storage as the clockwise node's storage. If the storage is empty it announces itself to all peers in the ring via *AnnounceJoiningNodeMsg*; otherwise, it makes a reed request via the *GetValueMsg* for each of the storage keys.
6. Once the read operation steps are carried out, the joining node receives *ReturnValueMsg*. If the item is not updated, it updates it by putting the new version and the new value. When the number of *ReturnValueMsg* is equal to the number of items in storage, it means that it has received read responses for each of the items, and therefore announces itself to all other peers via an *AnnounceJoiningNodeMsg*.
7. When peers receive an *AnnounceJoiningNodeMsg*, it updates the peer list and for each key in storage checks whether it is still responsible for the item or not and removes it if so.
8. If the join operation is not completed within 5000 ms, the timeout is triggered.

3.4 Leave Operation

1. The client sends a *LeaveRequestMsg* message to the node that has to leave the ring.
2. When the node receives the message, for each item it has in storage, it sends an *AddItemToStorageMsg* to the node that will have to insert the item into its storage. For each peer, it then sends an *AnnounceLeavingNodeMsg*.
3. When peers receive an *AnnounceLeavingNodeMsg*, the leaving node is removed from the node list.

3.5 Crash and Recovery Operation

1. The client sends a *CrashRequestMsg* message to the node that has to crash. The node that receives this message will enter the crash state and will not respond to any requests.
2. The client sends a *RecoveryRequestMsg* message to the node that has to recover, which contains the ID of a node from which to request the peer list.
3. The node that receives the *RecoveryRequestMsg* checks whether it is truly in a crash state, otherwise it sends an error message. If it is crashed, it creates a new recovery request and removes itself from the crash state, otherwise an error message is returned stating that the node was not crashed to begin with. It then sends a *GetPeerListMsg* to the node specified in the request and sets a timeout that will fire after 10000 ms.
4. The node sends the recovery node a *SendPeerListRecoveryMsg* containing the list of peers.
5. When the recovery node receives the list of peers, it creates a new storage in which it adds only the items for which it is still responsible. Then it finds the anti-clockwise neighbor to which it sends a *GetItemsListMsg*, to understand the new items for which it is responsible. We only ask the anti-clockwise neighbor because the only items we could possibly be missing in doing so are the ones for which the recovery node is the first holder, which could have not been created while the node was down because the request would have been queued and eliminated once the timeout would have been triggered.
6. The anti-clockwise neighbor sends a *SendItemsListRecoveryMsg*, containing its storage.

7. When the recovery node receives a *SendItemsListRecoveryMsg*, it looks at which of these items it needs to put into its storage and adds them. Finally, for each item in the storage it sends a read request via the *GetValueMsg* to update all the values.

8. Once the read operation steps are carried out, the recovery node receives *ReturnValueMsg*. If the item is not updated, it updates it by putting the new version and the new value.

4 Sequential Consistency

To guarantee sequential consistency two cases must be taken into account: write/write conflicts and write/read conflicts.
To avoid these two types of conflict we used the lock mechanism: each item, in addition to the key, value and version, has a value for the update lock and one for the read lock.
Every time an update is requested, the item is locked, and the item will not be available for other updates or reads until it is unlocked at the end of the operation.
If a read is requested, the read lock of the item will be increased, so as not to allow updating while it is being read but to allow there to be other reads; the update can only be done when the item is unlocked and the number of lock reads is equal to 0.
If the operations cannot be carried out because the item is not available, they will be queued.
Another choice was to include an additional data structure to guarantee consistency in the update operation, an array called *pendingRequests*. When, in the update operation, the quorum for the read operation is reached, the item is not unlocked, but the coordinator moves the request from *activeRequests* to *pendingRequests* and a *changeValueMsg* is sent to all nodes responsible for the item. A counter in the request keeps track of the number of ok messages are received by the coordinator until the quorum is reached. Only at this point is the item unlocked, guaranteeing that at least the majority of nodes have updated the value of the item before making it available once again for future operations.

5 Implementation

Some architectural choices are as follows:

- A crashed node simply does not respond to any type of message, unless it is a recovery request.
- The number of messages exchanged is minimal during read/write operations, as the coordinator of the operation sends messages only to the person responsible for that item (i.e., the first node that owns the item).
- The number of messages exchanged is also minimal during join/leave operations, since in the join the values are sent only to the nodes that will become responsible for them, and in leave the node sends the items only to the nodes that become the new responsible.
- The quorum for read and write has been set to

$$\frac{N}{2} + 1 \tag{1}$$

- There is a structure for each node, an array called *activeRequests*, that each node keeps containing the requests for which it acts as coordinator. Once a request is satisfied or the access to the item is denied (due to lock incompatibilities) the request is removed from the active requests.
- An interesting implementative choice regards the way requests are handled when the item is already locked in a way that the access to the item gets denied. In that case, instead of simply returning an error message, the request is added in a queue in the coordinator, *requestQueue*. This queue is periodically emptied thanks to a timeout, giving requests the chance to be re-submitted, hoping the unavailable item has been unlocked by then, before the request timeout clocks.