YSC4231: Parallel, Concurrent and Distributed Programming
Final Project

`AIM:` Applying my skills and knowledge of distributed programming using Scala actors to implement a *distributed, fault-tolerant replicated key-value store.*
`SRC:` tagged-release.

Overview of report structure

This report will chronologically discuss and describe the discoveries, design decisions, and related thoughts that contributed to my implementation of the distributed, fault-tolerant replicated key-value store. Excluding the introduction and conclusion, my discussion will be divided into six sections, each corresponding to a step from the six milestones suggested by the instruction:

1. Implement the standalone primary replica actor so that it correctly responds to key-value protocol messages without considering persistence or replication.
   [**Basic Primary Replica**]

2. Implement the secondary replica role so that it correctly responds to the `lookup` requests and accepts the replication protocol, without considering persistence.
   [**Basic Secondary Replica**]

3. Implement the replicator so that it correctly mediates between replication requests, snapshots, and acknowledgements.
   [**Replicator**]

4. Implement the use of persistence in the secondary replicas.
   [**Persistence for Secondary Replica**]

5. Implement the use of persistence and replication at the primary replica.
   [**Persistence and Replication for Primary Replica**]

6. Implement the sending of the initial state replication to newly joined replicas.
   [**Initial state replication**]

Introduction: Before Step 1

My biggest fear when beginning this project was that I would misunderstand or misinterpret a part of the instructions and waste a lot of time before realizing the mistake. It has happened before with some weekly assignments, and I did not want to repeat it for this final assignment. Hence, before jumping into Step 1, I invested a significant portion of my total time into understanding the ins-and-outs of this complex project. I initially read the instructions from the top to the bottom – taking my own notes along the way (included in `README.md`) – to gain a big-picture understanding of the distributed key-value store. Then, I selectively reread the sections that are applicable to each of the actors (and protocols) one-by-one, drawing mind maps with arrows to represent the messages required for each protocol. These notes and drawings, though far from perfect, proved to be incredibly helpful for reference as I progressed through the project. I believe this preparation was a critical and necessary step for my successful completion of this project.

Step 1: Basic Primary Replica

Step 1 was fairly simple to implement. Initially wondered where I should place the `mediator ! Join` operation but quickly noted that all instances of Replica must send the message immediately upon instantiation. Hence, I investigated what the equivalent of `__init__` of Python is for Scala classes and discovered that I can simply place it inside the class definition.

Moreover, I took some time in step 1 to get comfortable with the syntax and operations of Scala's built-in immutable map data structure to insert, remove, and lookup key-values in the store. I believe it was time well spent as maps are widely used in this project.

Additionally, I researched about the message delivery/ordering guarantees provided by Akka to consider whether I needed to implement anything to ensure the correct ordering of message by message id (no action was required on my part[1]).

Step 2: Basic Secondary Replica

Initially, I was confused as to how step 2 could be completed without working on step 3 (and 6) in tandem. In my mind, I couldn't imagine how a secondary replica actor's function could be tested with an incomplete replication protocol and without a initial state replication protocol. However, when I took a look at the testing script provided (which utilizes a replicator `TestProbe`), I quickly understood how it is possible to isolate the secondary replica node and test its functionalities individually. It made be rethink my (unit) testing approaches and how I should approach other large projects.

Beginning the implementation for the secondary replicas was initially slightly overwhelming. However, I found that it helps to think chronologically about the 'life-cycle' of a secondary replica – starting from when it joins and sends a message to the mediator to its subsequent interactions with its personal replicator actor. This helped me understand which features and functionalities I need to program for the secondary replica role.

Step 2 was when my scratch drawings (of the system) started to come in handy. In particular, I found myself getting confused by the id systems across the different protocols. After rereading parts of my notes and reviewing my sketches, I identified two id systems for the entire project and decided to consistently use `id` and `seq` to differentiate between the two ordering systems. Making this small change in my scripts allowed me to feel much more confident while navigating different protocols and messages.

Moreover, I implemented a simple `expected_seq` counter such that each secondary replica can keep track of the expected sequence number for `Snapshot` messages that it receives from its corresponding replicator. The counter is initialized to expect 0 as the first sequence number as the instructions clearly state that "each Replicator uses its own number sequence starting at zero."

I also used pattern matching on `valueOption` to identify if the `Snapshot` was carrying an `insert` or `remove` update operation, and handled them correspondingly.

Step 3: Replicator

When approaching this step, I was aware that there were two explicit cases to handle: `Replicate(key, valueOption, id)` and `SnapshotAck(key, seq)`. To implement the replicator's response to these messages, I first studied the variables that have been provided by the script (e.g. `acks`, `toCancel`, `_seqCounter`, etc...) and considered how I could best utilize them.

[1]https://doc.akka.io/docs/akka/2.2/general/message-delivery-guarantees.html

Let us first consider the response to receiving a message of type `Replicate(key, valueOption, id)`: Consulting my notes, I determined that the main response should be to create and send a `Snapshot` message to the replicator's partner replica. I noted that the instructions emphasized that the Replicator must make sure to periodically retransmit all unacknowledged changes to account for the case that either a `Snapshot` message or its corresponding `SnapshotAck` message is lost on the way. Hence, I jumped into the Akka documentation on Scheduler[2] to gain a better understanding of its methods `scheduler.schedule()` (and `scheduler.scheduleOnce()`). Considering the given auxiliary method `scheduleForPersistence` in `Replica.scala`, I adopted a similar structure to implement my own auxiliary method `scheduleForSnapshot`:

```scala
private def scheduleForSnapshot(k: String, vo: Option[String], seq: Long) {
    val cancellable =
      scheduler.schedule(Duration.Zero, 100 millis, replica, Snapshot(k, vo,
          seq))
    toCancel = toCancel + (seq -> cancellable)
}
```

This method takes a key (`k`), valueOption (`vo`), and sequence number (`seq`) and leverages `ActorSystem`'s `scheduler.schedule()` method to create a cancellable object which represents the sending of `Snapshot(k, vo, seq)` to the replicator's corresponding replica every 100 milliseconds starting from the moment the command is executed. Then, the sequence number of the Snapshot message and its associated cancellable object is added to the `toCancel` map which was conveniently given in the script. The purpose of the last step is to store this object somewhere accessible such that we can later on cancel the operation to stop sending the message. Utilizing `scheduler.schedule()` allows our replicator to handle multiple Snapshots in parallel.

Furthermore, I noted that once a `SnapshotAck` is received for a particular `seq`, I would require the original requester's ActorRef (i.e. `sender()`) and the `Replicate` message they sent to trigger the Snapshot with that particular `seq` such that the replicator can send back a `Replicated` message to indicate successful replication. The former (i.e. original requester's ActorRef) is likely to be the ActorRef of the primary replica (which is unchanging in our simplified key-value store) but in consideration for potential extensions to this key-value store where there may be multiple primary replicas or the primary replica is replaceable, it is wise to store the ActorRef of the sender for each `Replicate` message. Fortunately, the data structure to store this information was pre-built for us: `acks`. Putting all of this together, the handling of `Replicate(key, valueOption, id)` was implemented as follows:

```scala
case Replicate(key, valueOption, id) =>
  val seq = nextSeq
  acks += (seq -> (sender(), Replicate(key, valueOption, id)))
  scheduleForSnapshot(key, valueOption, seq)
```

Handling `SnapshotAck(key, seq)` was relatively simple thanks to the data structures described above. Once a `SnapshotAck` was received, the replicator checks the requester's ActorRef and original message by looking up the `seq` in `acks`. If it doesn't exist, it is likely a duplicate/delayed acknowledgement so we do nothing. If it does exist (i.e. first

[2]https://doc.akka.io/docs/akka/2.4/scala/scheduler.html

acknowledgement for the interested Snapshot), we cancel the associated cancellable object to stop the sending of redundant Snapshot messages, delete the `seq` and its associated value from `toCancel`, and then send the appropriate Replicated message back to the original sender. Hence, the case for `SnapshotAck(key, seq)` is handled by the replicator as follows:

```
case SnapshotAck(key, seq) =>
acks.get(seq) match {
    case Some((sdr, Replicate(k, v0, id))) =>
      toCancel(seq).cancel()
      toCancel -= seq
      sdr ! Replicated(k, id)
}
```

Step 4: Persistence for Secondary Replica

An epiphany that I had while implementing step 4 is that there is no `OperationFailure` equivalent message for the secondary replicas to send in response to `Snapshot`. In other words, unlike the primary replica whose response to a (update) request can either be a success or failure, the secondary replica always succeeds to reply to every distinct `Snapshot` message with a `SnapshotAck` *eventually*. The associated primary replica may 'fail' its request due to the slowness of the secondary replica, but if we isolate the logic of the secondary replica, the secondary replica will keep trying for as long it takes until the update is persisted to send a `SnapshotAck` in response to every `Snapshot` message. Although this seems like a self-evident, obvious fact in retrospect (perhaps I should have read the instructions an extra time), I initially struggled considering whether I should set an upper bound time limit for the responding to each `Snapshot` message. After this epiphany however, step 4 became a lot simpler.

The modification I made to the secondary replica's handling of `Snapshot` is calling `scheduleForPersistence(key, valueOption, seq)` instead of immediately sending `SnapshotAck`. Hence, by this new structure, a `SnapshotAck` message is sent in response to a `Snapshot` only when a `Persisted` message acknowledges the persistence of the interested update.

```
case Snapshot(key: String, valueOption: Option[String], seq: Long) =>
  if (seq < expected_seq) sender() ! SnapshotAck(key, seq)
  else if (seq > expected_seq) {}
  else {
    valueOption match {
      case None => kv -= key
      case Some(value) => kv += (key -> value)
    }
    scheduleForPersistence(key, valueOption, seq)
    expected_seq += 1
  }
```

`scheduleForPersistence()` was provided in the script and is implemented as follows:

```
private def scheduleForPersistence(k: String, vo: Option[String], seq: Long) {
    val scheduler = context.system.scheduler
    val cancellable =
      scheduler.schedule(Duration.Zero, 100 millis, persistence, Persist(k,
          vo, seq))
    toCancelPers = toCancelPers + (seq -> (sender, cancellable))
}
```

Similar to `scheduleForSnapshot()`, `scheduleForPersistence()` creates a cancellable object and stores the object in a shared data structure (i.e. `toCancelPers`) such that it can be accessed and cancelled in the future. For this method, the operation that is being repeated every 100 millisecond from the moment the cancellable is created is sending `Persist(k, vo, seq)` to the persistence actor associated with the interested secondary replica.

One modification I have made to the given `scheduleForSnapshot()` is to use the sequence number as identification number for the persistence protocol. The original code provided for `scheduleForSnapshot()` used the variable `id` instead of `seq`, which initially made me consider using the original update request id. However, I decided against it. The `id` associated with the update request (to the primary replica) less readily available for the secondary replicas and needlessly adds complexity to the ordering system required for the persistence protocol. Moreover, pattern matching is simplified in handling `Persisted` (i.e. no need to store original `Snapshot` message).

In addition, the secondary replica will receive `Persisted(key, seq)` messages from its associated persistence actor as acknowledgement that the update (identified by `seq`) is persisted. Similar to how replicator handles `SnapshotAck(key, seq)`, the second replica will first lookup the sender and cancellable associated with the `seq` in `toCancelPers`. If it doesn't exist, the `Persisted` message is probably a duplicate (which arrived later). If it does exist, the cancellable object is cancelled and removed from `toCancelPers`, and a `SnapshotAck` message is sent to the sender of the interested `Snapshot` message (i.e. its associated Replicator):

```
case Persisted(key, seq) =>
  toCancelPers.get(seq) match {
    case Some((sdr, cancellable)) =>
      cancellable.cancel()
      toCancelPers -= seq
      sdr ! SnapshotAck(key, seq)
  }
```

Step 5: Persistence and Replication for Primary Replica

To implement persistence and replication in the primary replica actor, several new data structures were introduced which are briefly described in the code snippet below:

```
// Conditions for OperationAck
// 1. PERSISTENCE at current (i.e. primary) replica
// Set[ID] of updates that are not yet persisted [PersAck ~ Persisted]
var persAcks_due = Set.empty[Long]

// 2. All sec replicas have ACKed the REPLICATION of update
// (ID -> number of repAcks due/outstanding) [repAck ~ Replicated]
var num_repAcks_due_by_id = Map.empty[Long, Int]
// (ID -> requester's ActorRef)
var sender_by_id = Map.empty[Long, ActorRef]
```

Let us first consider how the handling of the update messages (i.e. `Insert/Remove`) were modified to provide replication and persistence. For update messages, the largest change I have undertaken is deferring the sending of `OperationAck` to when its corresponding `Persisted` message or appropriate `Replicated` message is received. This is similar to the logic from step 5. Considering the two conditions that are required for the sending of `OperationAck` (described in the comments in the above code snippet), I decided that the `OperationAck` should be sent when the message that finally satisfies the conditions is received by the primary replica. If it took longer to persist the update than replicate it to all of its secondary replicas, this will be when the `Persisted` message arrives. If it is vice-versa, this will be when the last outstanding `Replicated` message arrives.

For handling `Insert/Remove`, the method `scheduleForPersistence` is utilized for the same reason as in step 5. In addition, the primary replica checks if there are any secondary replicators to whom it must replicate this update. If there is, we add the update request id and the number of replicators (i.e. secondary replicas) at the moment to a shared map variable called `num_repAcks_due_by_id` which is used to track how many outstanding `Replicated` messages remain. Then, the appropriate `Replicated` message is sent to each of the replicators.

In addition, we must implement a way to send an `OperationFailure` message to the requester if the conditions listed above are not met within 1 second since receiving the update request message. To do this, I used `scheduler.scheduleOnce(1 second)`{} which allows me to schedule an operation that checks if the two conditions for `OperationAck` have been met after 1 second. My implementation of handling `Persisted` and `Replicated` messages involve deleting the request id from `persAcks_due` once the update has been persisted and also from `num_repAcks_due_by_id` once all outstanding `Replicated` messages have been received (i.e. repAcks due = 0). Hence, the operation to determine whether the node should send a `OperationFailure` is as simple as:

```
scheduler.scheduleOnce(1 second) {
    if (num_repAcks_due_by_id.contains(id) || persAcks_due.contains(id))
        sender_by_id(id) ! OperationFailure(id)
}
```

Note how `sender_by_id` map comes in handy in this situation. Initially, I naively tried using `sender()` instead but came across a `deadLetters` error. After investigating the nature of cancellable objects and dead letter errors, I realized that a `sender_by_id` would be necessary. The primary replica appends a key-value pair into `sender_by_id` for every `Insert/Remove` request.

The handling of `Persisted` in the primary replica is almost identical to that in secondary replicas with only a few slight differences: The `id` associated with the `Persisted` message is removed from `persAcks_due`. Then, in addition, the replica checks if the map `num_repAcks_due_by_id` contains the key `id`: if it doesn't, then the two conditions have been fulfilled so an `OperationAck` message for that `id` is sent to the original requester.

```
case Persisted(key, id) =>
  toCancelPers.get(id) match {
    case Some((sdr, cancellable)) =>
      cancellable.cancel()
      toCancelPers -= id
      persAcks_due -= id
      if (!num_repAcks_due_by_id.contains(id)) sdr ! OperationAck(id)
  }
```

To handle `Replicated`, the primary replica first decrements the outstanding repAck count for that id in `num_repAcks_due_by_id`. If the updated outstanding repAck count is not equal to 0 (i.e. more repAck pending), only `num_repAcks_due_by_id` is updated and no further actions are taken. However, if there are no more pending `Replicated` messages for the given update request id (repAck due=0), then the request id (and its corresponding value) is deleted from `num_repAcks_due_by_id`. Then, the replica checks if the update has been persisted by checking if `id` is not contained in `persAcks_due`: if so, an `OperationAck` message for that `id` is sent to the original requester.

```
case Replicated(key: String, id: Long) =>
    repAcks_due_by_replicator += (sender() ->
        (repAcks_due_by_replicator(sender())-id))
    val outstanding_replicatedAck = num_repAcks_due_by_id(id) - 1
    if (outstanding_replicatedAck != 0) {
      num_repAcks_due_by_id += (id -> outstanding_replicatedAck)
    } else {
      num_repAcks_due_by_id -= id
      if (!persAcks_due.contains(id)) sender_by_id(id) ! OperationAck(id)
    }
```

Note how the map `sender_by_id` is once again useful for fetching the ActorRef of the original requester.

Step 6: Initial state replication

To implement initial state replication and the appropriate response to newly joining or leaving secondary replicas, a few more variables are introduced:

```
// (KEY -> ID) where ID=most recent update id for that key
var ki = Map.empty[String, Long]
// (Replicator -> Set[ID])
var repAcks_due_by_replicator = Map.empty[ActorRef, Set[Long]]
```

The problem that I was faced with when trying to implement the initial state replication protocol was that the nature of this protocol is sufficiently different from a regular replication protocol (e.g. only replicating to one replica) that I struggled to find a way to integrate the two processes together. I initially even considered creating a new message such as `ReplicateInit` which could be handled in a completely different way. However, after consideration and experimentation, I turned my head towards using the existing `Replicate` message. Then, the question of `id` arose – the initial state replication must not overwrite future update replications and must be ordered correctly in relation to other replication updates. Should I use negative numbers for `id`? Should I keep track of the minimum update request id and only use id's below the minimum? I considered several options but ultimately decided to use the original update id for each of the key-value pairs that are being transferred over. This will allow proper ordering of replication events that is consistent with the key-value store at the primary replica. Hence, `ki` was created with the purpose of remembering the original insert request ids of the keys that are present in the key-value store. `ki` is updated every time a new `Insert` request is received.

`repAcks_due_by_replicator` is a map that associates a replicator's ActorRef to the set of id's that it has yet to send a `Replicated` message for. This map is updated by `Insert/Remove` messages (i.e. add id to `repAcks_due_by_replicator(r)`) and `Replicated` messages (i.e. remove id from `repAcks_due_by_replicator(r)`).

Let us consider how the `Replicas(replicas)` message is handled by the primary replica: the primary replica uses set subtraction to determine a. the updated set of secondary replicas, b. set of newly joined secondary replicas, and c. set of secondary replicas that have left (i.e. Set[ActorRef]). The primary replica first considers the newly joined replicas. Iterating through each newly joined replica (type: `ActorRef`), the primary replica creates a new replicator that is associated with the new replica, adds the ActorRef of the new replicator to the set of replicators, updates the `secondaries` map, adds the new replicator's ActorRef and an empty set as key-value pair in `repAcks_due_by_replicator` and begins the initial state replication. The initial state replication protocol is simply iterating through the key-value store of the primary replica and sending a `Replicate` message to the new replicator. As stated previously, `ki` is utilized here to fetch the original update request id for each key to ensure that the initial state does not overwrite any future update replications.

As a side-note: To handle the resulting series of `Replicated` messages whose message id's cause an error to be thrown when executing `num_repAcks_due_by_id(id)` (as these are id's have already removed from `num_repAcks_due_by_id` when its replication protocol succeeded), I decided to wrap the `Replicated` message handling with an if-statement:

```
case Replicated(key: String, id: Long) =>
  if (num_repAcks_due_by_id.contains(id)) {...}
```

By doing this, the `Replicated` messages caused by the initial state replication (that may throw errors) are ignored by the primary replica.

Returning to the primary replica's handling of the `Replicas(replicas)` message, the primary replica then proceeds to consider any replicas that have left the distributed key-value store network. Then, it iterates through each of their ActorRefs, determines the ActorRef of their corresponding replicators using `secondaries` and removes it from the set of `replicators`. Furthermore, as the instructions suggested, the primary replica waives the outstanding `Replicated` acknowledgement messages of the leaving secondary replica (and its corresponding replicator) by fetching the set of update request id's for which the leaving replica(tor) has yet to send a `Replicated` message. Then, the primary replica sends a `Replicated` message with for each of the id's in the set to itself (note: content of `key` included inside the `Replicated` message is irrelevant in the handling of the message, so I set it to "k" – can be any string). Furthermore, the replicator is removed from `repAcks_due_by_replicator` (along with its corresponding value) before, finally being terminated with a `PoisonPill`. From the instructions, I gathered that the secondary replica leaving does not implicate in the termination of the leaving replica itself (but only its replicator), hence the replica is not terminated.

```scala
case Replicas(replicas) =>
  val updated_sec_replicas = replicas - self

  val new_sec_replicas: Set[ActorRef] = updated_sec_replicas --
     secondaries.keySet
  new_sec_replicas.foreach { new_sec_replica =>
    val new_replicator = context.actorOf(Replicator.props(new_sec_replica))
    replicators += new_replicator
    secondaries += (new_sec_replica -> new_replicator)
    repAcks_due_by_replicator += (new_replicator -> Set.empty[Long])
    kv.foreach { key_val =>
      new_replicator ! Replicate(key_val._1, Some(key_val._2), ki(key_val._1))
    }
  }

  val removed_sec_replicas: Set[ActorRef] = secondaries.keySet --
     updated_sec_replicas
  removed_sec_replicas.foreach { removed_sec_replica =>
    val corresponding_replicator = secondaries(removed_sec_replica)
    replicators -= corresponding_replicator
    repAcks_due_by_replicator(corresponding_replicator).foreach(id => self !
       Replicated("k",id))
    repAcks_due_by_replicator -= corresponding_replicator
    corresponding_replicator ! PoisonPill
 }
```

Given that `sender()` is called in the handling of `Replicated` message in order to update `repAcks_due_by_replicator`, I added another if condition such that messages sent by the `self` (i.e. the primary replica) does not update `repAcks_due_by_replicator`. This is acceptable as the leaving replica's corresponding replicator (and its value pair) is removed from `repAcks_due_by_replicator` afterwards anyways. The primary replica's modified handling of `Replicated` message is as follows:

```
case Replicated(key: String, id: Long) =>
  if (num_repAcks_due_by_id.contains(id)) {
    if (sender() != self) {
      repAcks_due_by_replicator += (sender() ->
          (repAcks_due_by_replicator(sender())-id))
    }
    val outstanding_replicatedAck = num_repAcks_due_by_id(id) - 1
    if (outstanding_replicatedAck != 0) {
      num_repAcks_due_by_id += (id -> outstanding_replicatedAck)
    } else {
      num_repAcks_due_by_id -= id
      if (!persAcks_due.contains(id)) sender_by_id(id) ! OperationAck(id)
    }
  }
```

That concludes the journey of my implementation of the distributed, fault-tolerant replicated key-value store. For the source code, please refer to the tagged GitHub release accessible via the hyperlink on page 1.

Conclusion

This was definitely one of the most complex and fascinating projects that I have implemented in my undergraduate courses so far. Through this project, I have gained a better understanding of how a computer science project should be approached, developed, and tested. I have grown a strong interest for the discipline of distributed programming which I hope to further explore.