Project of Distributed Algorithms: Robust Key-Value Store

Romane Thoirey

Maximilien Frat Leprince

Bixente Grandjean

Index:

Introduction:

Context:

In the distributed algorithms, we have been given a project for a three members team.

The project:

The goal of this project is to get an initial experience in designing a fault-tolerant distributed system.

We give a key-value store implementation for an environment of N asynchronous processes with distinct identifiers publicly known. Every two processes can communicate via a reliable asynchronous point-to-point channel. Up to f < N/2 of the processes are subject to crash failures: a faulty process prematurely stops taking steps of its algorithm. A process that never crashes is called correct. The implementation ensures that in its every execution, the following condition as are met:

-Safety: The corresponding history is linearizable with respect to the sequential specification above

-Liveness: Every operation invoked by a correct process eventually returns. We will use Java to fulfil the project.

Technology:

We used Java with the AKKA environment.

High level description of the system

The system

The system is a set of multi-writer and multi-reader atomic registers of multiples protocols.

- The server protocol: in which each replica receives read and write requests. If the tag enclosed in the received message is higher than the local tag of the replica, the local copy of the replica is updated and then a reply is sent to the requester with an acknowledgment and the local copy.

- The write protocol: the writer sends messages to all the servers and waits until it receives replies from a quorum of servers. Then, the writer chooses the pair with the highest tag among all the replies received, generating a new tag by incrementing the timestamp inside the maximum chosen tag, propagating the maximum tag-value pair to a quorum of servers.

- The read protocol: the reader sends messages to all the servers and waits until it receives replies from a quorum of servers. Then, the reader chooses the pair with the highest tag among all the replies received, propagating the maximum tag-value pair to a quorum of servers.

The processes:

We create a list of actors representing the number of processes which execute the different actions. Using a shuffle function, we set the first half (+1 if the number is odd) processes as corrects and the last half of processes as faulty, then we launch the processes.

In our program, we implemented multiple five different states:

- **Correct**: if the process that we created is correct
- **Crashed**: if the process is faulty
- Two specific waiting states, **Wait_Read** for the read waiting and **Wait_Write** for the write waiting. The goal is to avoid any switch from writing to reading (or the opposite) during the waiting
- **Correct_wait**: the second correct state which is created to indicate that the process has finished his put operations and will now do his get operations

Local copies of the values are stored with the timestamp value and the process identifier, to be able to recognize the process easily. There are WriteRequest, ReadRequest, WriteResponse, and ReadResponse functions, the Write is always before the Read. For the writing, a request is sent, then a response is sent back with the content to write. For the reading, a request is sent, then a response is sent back with what must be read.

The operations:

There are three states of operation:

- 0: no operation active
- 1: get operation
- 2: put operation

The onReceive function:

The onReceive function check for faulty processes, then the other processes launch put and get operations.

- <u>If the message received is a ReadRequest:</u> a ReadResponse is launched with its own timestamp.

- <u>If the message received is a WriteRequest and t’ > t or t’==t and p’>p:</u> we update the values of v and t the timestamp, then we send the WriteResponse to the process.

- <u>If the message received is a ReadResponse and the process is waiting for a read state and P1==P0:</u> the expected read response is received. If more than half responses have been received, we change the state and send a new request. We compare the values by taking the highest timestamp or, if the timestamp is equal, taking the highest process identifier, and we update them. We check for put and get operations to perform and launch them if they are active.

- <u>If the message received is a WriteResponse and the process is in a waiting for a write state and P1==P0:</u> the expected write response is received. If more than half responses have been received, we change the state and complete the put operation. If the putCounter is different than M, we increment the putCounter, set the active operation to 0 and launch the get statement. If the active operation is a get, we complete it and if the getCounter is not equal to M, we increment it, set the active operation to 0.

Pseudocode of the implementation

Main.java file

```
Create integer system size, system actor, a Date, Arraylist of actor references

try
    for i 0→system size
        Create a new process and put it in the arraylist of actor references
    Print the date of system creation
    Shuffle members arraylist
    Put a state for each process, with half active and half faulty
catch (IOExeption ioe)
finally
close the system
```

Process.java file

```
Class Process extends UntypedAbstractActor
    Create a log
Create Props method (id of the process)
    Return the created Process

Create the Members constructor (arraylist of type actorRef)

Create a state class
    Create state integer

Create the state constructor
```

Create a StampedValue class

    Create value integer, seqnum integer, pid integer

    Create the StampedValue constructor

Create a WriteRequest class

    Create StampedValue, localseqnum integer

    Create the WriteRequest constructor

Create a WriteResponse class

    Create a localseqnum integer

    Create the WriteResponse constructor

Create ReadRequest class

    Create a localseqnum integer

    Create the ReadRequest constructor

Create a ReadResponse class

    Create StampedValue, localseqnum integer

    Create the ReadResponse constructor

Create id integer

Create arraylist of type Members correspondig to the members known to the process

Create state integer which is the process state

Create an activeop integer which is the currently executed operation: 0 no operation, 2 put operation, 3 get operation

Create a stampedValue : local sequence number (number of operations performed)

```
Create an Arraylist : stores messages received in a write or read
Create a M integer: the number of put and get operation to perform
Create a putCounter integer: all the put operation performed
Create a getCounter integer: all the get operation performed
Create the Process constructor

Create putOperation method
        Add 1 to localseqnum
        Print that we use put operation and its value
        Put the activeop at 2, that say the active operation is a put
        Delete the current arraylist of messages
        Send a ReadRequest to all the other processes
        Put the current state to Wait_Read, that tells us that the process is on a read waiting
state.

Create getOperation method
        Add 1 to localseqnum
        Print that we use get operation and its value
        Put the activeop at 3, that say the active operation is a get
        Delete the current arraylist of messages
        Send a ReadRequest to all the other processes
        Put the current state to Wait-Read, that tells us that the process is on a read waiting
state.
Create an override onReceive method throws Exception (arraylist of messages)
        if the current state is not crashed (not faulty)
                Create an actorRef that takes the current reference of the process
        if messages is on the members arraylist
                mem of the process takes the members messages
        else
        if messages is a State
                if the state is Correct
```

```
            Start the putOperation method
        if the state is Correct_Read
            Start the getOperation method
        else
            Print that the process is faulty
    else
    if messages is a ReadRequest
        Print that the process has received a read request and from who
        Send a ReadResponse to the actor which sent this ReadRequest
    else
    if messages is a WriteRequest
        if the seqnum value and the pid value > the value of this current process
            Replace the current value by the biggest one
            Print the update of the timestamp
            Send a WriteResponse to the actor that sent this ReadResponse
    else
    if messages is a ReadResponse and the process is on the state Wait_Read
        Print that it has received a read response and from who
        Add the message to the Arraylist
        if it received most of the read response
            Print that it has received a minimum of read responses
            Put its state to Correct, because it is no more on waiting state
            if the seqnum value and the pid value of each readResponse it
        received is over the value of this current process
                Replace the current value by the biggest one
        Print the new timestamp
        if the activeop is 2
            Put the state as write waiting state
            Clear the arraylist messages
            Send to all process a writeRequest with the value seqnum+1
        if the activeop is 3
            Put the state as write waiting state
            Clear the arraylist messages
            Send to all process a writeRequest with its own stampedvalue
    else
    if messages is a WriteResponse and state Wait_Write
        Print that it has received a write response and from who
        Add the message to the arraylist
        if it received most of the read response
            if the activeop is 2
                Print that we have received a minimum of write responses
                Put state to Correct, no waiting anymore
                Print that the put operation is done
                if the M put operation are done
                    Switch the state, and start a new get operation
                else
                    add 1 to the putCounter
                    Switch the state, and start a new put operation
            if the activeop is 3
                Print that we have received a quorum of write responses
                Put state to Correct, no waiting anymore
                Print that the put operation is done
                if the M put operation are done
                    Put the activeop to 0, no more operation to do
                    Put the process's state to 0, the process have done all
                its tasks.
                else
                    add 1 to the putCounter
                    Switch the state, and start a new put operation
            else unhandled the message
```

Proof of Correctness

To see if there is a proof a correctness, we must show that both Safety and Liveness properties are respected.

- First, <u>Safety property</u> says that “Something bad never happens”. In this case, it means that the server never receives an unacceptable state.

- <u>Liveness property</u> says that “Something eventually good happens”. The server eventually enters in a wanted state, like “CORRECT” state for example.

In our program, there are no deadlocks, since it never enters in a state which no longer progress. If it can’t proceed an operation, the state is updated to faulty and it’s stopped to launch a new process.

The mutual exclusion is when two process are never in their critical section at the same time. The mutual exclusion property is verified in the program. In fact, each reading action read the last written value, it avoids this problem.

Also, with the key story, we can establish a chronology of the different steps for each process, even if we are in a concurrent system. All of this proves the safety properties.

In our concurrent system, with the Liveness property, a process must always send something back, even if there is a crash. All operations of the program (put/get, read/write) return something back, eventually good. So, we can say that Liveness property is verified.

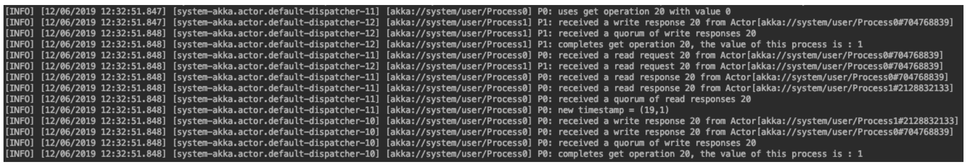
Both safety and liveness properties are true in our concurrent program, so this is a proof of correctness.

Performance analysis

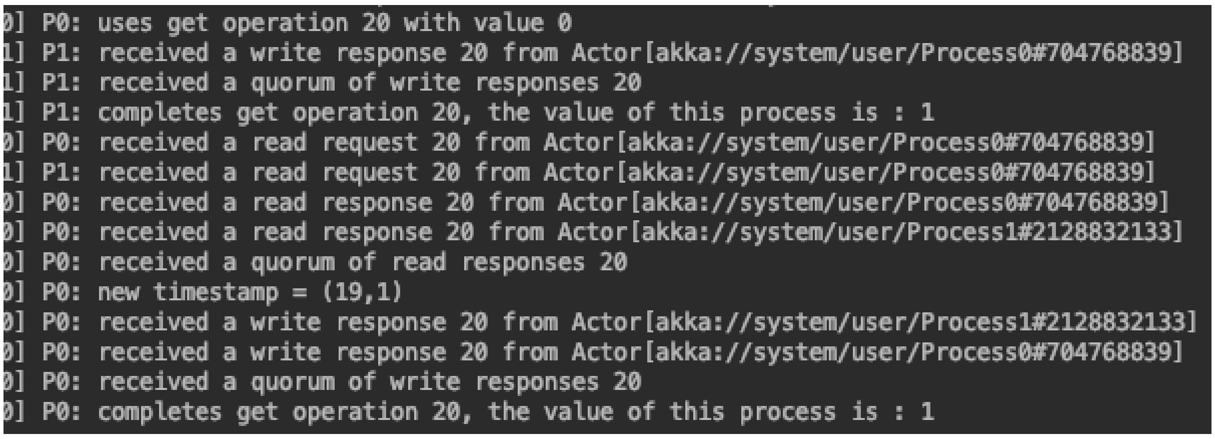
We test our program on different N_PROCESS and M_OPERATIONS to see the state of our server.

M N	3	10	100
3	0.41s	1.50s	2.50s
10	1.50s	1.50s	3.00s
100	2.50s	2.50s	7.00s

This is an example of output that we have with N_PROCESS = 3 and M_OPERATION = 10.



```
[INFO] [12/06/2019 12:32:51.847] [system-akka.actor.default-dispatcher-11] [akka://system/user/Process0] P0: uses get operation 20 with value 0
[INFO] [12/06/2019 12:32:51.847] [system-akka.actor.default-dispatcher-12] [akka://system/user/Process1] P1: received a write response 20 from Actor[akka://system/user/Process0#704768839]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-12] [akka://system/user/Process1] P1: received a quorum of write responses 20
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-12] [akka://system/user/Process1] P1: completes get operation 20, the value of this process is : 1
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-11] [akka://system/user/Process0] P0: received a read request 20 from Actor[akka://system/user/Process0#704768839]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-12] [akka://system/user/Process1] P1: received a read request 20 from Actor[akka://system/user/Process0#704768839]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-11] [akka://system/user/Process0] P0: received a read response 20 from Actor[akka://system/user/Process0#704768839]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-11] [akka://system/user/Process0] P0: received a read response 20 from Actor[akka://system/user/Process1#2128832133]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-11] [akka://system/user/Process0] P0: received a quorum of read responses 20
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-11] [akka://system/user/Process0] P0: new timestamp = (19,1)
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-10] [akka://system/user/Process0] P0: received a write response 20 from Actor[akka://system/user/Process1#2128832133]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-10] [akka://system/user/Process0] P0: received a write response 20 from Actor[akka://system/user/Process0#704768839]
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-10] [akka://system/user/Process0] P0: received a quorum of write responses 20
[INFO] [12/06/2019 12:32:51.848] [system-akka.actor.default-dispatcher-10] [akka://system/user/Process0] P0: completes get operation 20, the value of this process is : 1
```



```
0] P0: uses get operation 20 with value 0
1] P1: received a write response 20 from Actor[akka://system/user/Process0#704768839]
1] P1: received a quorum of write responses 20
1] P1: completes get operation 20, the value of this process is : 1
0] P0: received a read request 20 from Actor[akka://system/user/Process0#704768839]
1] P1: received a read request 20 from Actor[akka://system/user/Process0#704768839]
0] P0: received a read response 20 from Actor[akka://system/user/Process0#704768839]
0] P0: received a read response 20 from Actor[akka://system/user/Process1#2128832133]
0] P0: received a quorum of read responses 20
0] P0: new timestamp = (19,1)
0] P0: received a write response 20 from Actor[akka://system/user/Process1#2128832133]
0] P0: received a write response 20 from Actor[akka://system/user/Process0#704768839]
0] P0: received a quorum of write responses 20
0] P0: completes get operation 20, the value of this process is : 1
```

Bibliography

https://developer.lightbend.com/guides/akka-quickstart-java/

https://perso.telecom-paristech.fr/kuznetso/EFREI19/project.pdf

https://gitlab.telecom-paris.fr/petr.kuznetsov/slr210-projects/tree/master?fbclid=IwAR3E6klyaYSVAojkUr5SN71Y75DL-giGssm2BarDQJFnAEfKuuioYE5tiGg