2nd Phase Report

Daniel Flamino nº45465, Diogo Silvério nº45679 e Rita Macedo nº46033

Algoritmos e Sistemas Distribuídos

Introduction

For this project our main objective was to implement a replicated key value storage service using state machine replication. The state machine replication layer uses Multi Paxos. In the first chapter we explain how the system is organized. Next we explain more deeply each of the layers organization and in the third chapter we present the pseudo code or each layer. In the end we explain how we did the experimental evaluation and analyze the results.

1 Overview

The Project was implemented in the Akka/Scala ecosystem. Three layers for the service were implemented each one representing an actor on the system:

- The Application Layer that deals with the request made by the clients and the answers that are given back to him.
- The State Machine that contains a ordered list of the operations that are supposed to be executed in the system, as well as the actual key value store.
- The Multi Paxos decides which operation is going to be executed in each round of the system.

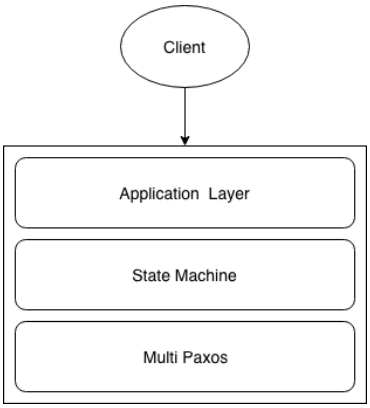

Figure 1: Representation of a Replica and a client in the System

Each replica of the system contains one instance of each of the three layers. Each layer communicates with the layer above and under it, and with the same corresponding layer of the other replicas. We also implement the clients that can communicate with any of the replicas.

2 Detailed Overview

2.1 Client

The client is not a layer of a replica of the system. It has an independent IP and port and can communicate with any of the replicas in the system. Mainly it communicates with the top layer of the replica but

it also communicates directly to get an updated list of replicas with whom it can communicate.

The client maintains a list of all the operations he has requested but has not received an answer yet.

When the client wants to request an operation he puts the operation on the list of operation and, if there is only one operation on the list, he sends the operation to the Application Layer of a random replica.

If the client receives a result then it takes the operation correspondent to that result from the list of waiting operations and if the list still has operations in it then it sends a ClientOp to the Application Layer of a random replica.

If the client receives a timeout it means that it didn't receive the answer that he needed in the determined time so he re-sends a request ClientOp to a different replica.

2.2 Application Layer

This layer is the one that for most part interacts directly with the clients. This layer has two main objectives: receive the client messages with the operation to be done and return the answer when the operation is completed.

In this layer we have a map to organize every operation that has already been done and a list to organize the requests made directly from a client to that replica.

When this layer receives a message from a client it verifies if for some reason the request has already been made and completed, if it has then it sends the answer to the client, if it hasn't then it sends a message with the operation requested to the lower level - the state machine - to be organized and concluded.

When this layer receives a message from a state machine, the answer is inserted in the map with all the answers and it is verified if the operation was a direct request for that replica, if it is the answer is given to the client.

2.3 State Machine

This layer has the objective of maintaining the history of the system, meaning it maintains an ordered list with all the operations made in the system.

This layer has an array with the state history, the IP and port of the leader and a Queue with the operations requested by the clients that are waiting to be decided.

In this layer there are three important operations - LeaderChange, Decided and OrderOperation.

The operation LeaderChange is requested by the bottom layer, the multi paxos, when a leader has died and another has been put in its place, so the state machine always has the updated leader.

The operation OrderOperation is called by the top layer, the Application Layer, to request another operation asked by a client. When this operation is requested it is verified if the leader is null or it is the leader. If that is the case then the operation is put in the operationQueue and if the queue only has one operation a propose is called on the bottom layer. The main idea is that only one operation is requested to the leader at a time, so the new operation is requested only if the paxos doesn't already have an operation being processed. If this replica is not the leader an OrderOperation is called on the state machine of the leader.

The operation Decided is called by the bottom layer when a new operation is decided. In this point it is verified which is the next operation to be executed on the stateHistory and from that point until there is an empty space on the array the operations are executed and an answer is sent to the top layer. After that the current round is incremented and if there is still operations on the queue, if that replica is the leader it sends a new propose with the operation on the head of the queue to the bottom layer, if the replica is not the leader it requests the OrderOperation to the state machine of the leader.

2.4 Multi-Paxos

The main objective of this layer is to order the operations requested by the clients, meaning that it decides which is the order in which the operations are going to be performed. It does this first by deciding a leader. After deciding a leader he is the only one that can send an accept to all the replicas to ask them to accept the new operation. When the majority decides that the operation is ok, the decided operation is communicated to the top layer in every replica.

There are two operations to change the leader, Prepare and PrepareOk and two methods to accept an operation, Accept and AcceptOk. The method Propose is always called from the top layer, the state machine, and it is only called if that replica is the leader or if there is no leader.

The method AddReplica is used if a client asks to add a replica and the method RemoveReplica is used if a client asks to remove a replica or if a replica sees that the leader has died.

When a value is proposed the replica verifies if there is as leader, if there is not she sends a Propose (for that replica to be the leader) to every replica in the system and sets a timeout for receiving the acceptances of its proposition, if she is the leader she sends an Accept to every replica in the system for them to accept the new operation and sets a timeout to receive an AcceptOk from the replicas.

When a replica receives a Prepare she verifies if the sequence number from the proposed leader is bigger than the one he's seen before. If it is, this layer tells the top layer (state machine) that the leader has changed and sends the leader a PrepareOk.

If the proposed leader receives a PrepareOk, he adds it to its list of PrepareOk received, and verifies if it already has a majority of answers. After that, he finds on the list the operation corresponding to the highest sequence number, if that operation exists he asks every replica to accept that operation, if that operation is null it means that there was not a replica that has accepted an operation before, for that reason he can ask for acceptance on his own operation. After sending the Accept to every replica with the decided operation he sets a timer to wait for the answer to see if the operation is accepted.

When an Accept is received, after updating the round number (if necessary) the replica verifies if the sequence number received is bigger than the sequence number that she has seen. If it is she updates her sequence number and her operation for the one received and sends an AcceptOk for the leader.

Finally, when a replica receives an AcceptOk she checks if the sequence number received is bigger or equal to the one she has, if it is bigger she updates her sequence number, her operation and restarts the counter of AcceptOk received, if it is equal she increments the AcceptOk counter and verifies if she already has a majority. If she has majority she restarts the counter and sends a Decided to the top layer, the state machine, with the operation decided.

3 Pseudo Code

3.1 Client

Algorithm 1: `Client (part 1)`

Interface:
Requests:
Read(key)
Write(key)
Add(replica)
Remove(replica)
DeliverResult(op, result)
OperationTimeout(oldReplicaIpPort, op)
DeliverResult(op, result)
Indications:
ClientOp (op)
AskForReplica()

State:
myIpPort
knownReplicas //set of known replicas
opsToExecute //List of operations that the client is waiting for the reply
Upon Init(myHostname, myPort) do:
$myIpPort \leftarrow myHostname + " : " + myPort$;
$knownReplicas \leftarrow \{\}$
$opsToExecute \leftarrow \{\}$;

Procedure chooseReplica() do:
return getRandomFrom(knownReplicas);

Algorithm 2: `Client (part 2)`

Procedure sendOperation (op) do:
$opsToExecute \leftarrow opsToExecute \cup op$;
  if $|opsToExecute| == 1$ **then**
    $replicaIpPort \leftarrow chooseReplica()$;
    Trigger ClientOp(op);
    Setup Timer OperationTimeout (1000, replicaIpPort, op);
  end if

Upon Read (key) do:
$op \leftarrow (myIpPort + \text{" :}$
$\text{"} + start.toString + \text{"»} + Random(), myIpPort, \text{"}read\text{"}, key, null)$;
sendOperation (op);

Upon Write (key) do:
$op \leftarrow (myIpPort + \text{" :}$
$\text{"} + start.toString + \text{"»} + Random(), myIpPort, \text{"}write\text{"}, key, null)$;
sendOperation (op);

Upon Add (replica) do:
$op \leftarrow (myIpPort + \text{" :}$
$\text{"} + start.toString + \text{"»} + Random(), myIpPort, \text{"}add\text{"}, replica, null)$;
sendOperation (op);

Upon Remove (replica) do:
$op \leftarrow (myIpPort + \text{" :}$
$\text{"} + start.toString + \text{"»} + Random(), myIpPort, \text{"}remove\text{"}, replica, null)$;
sendOperation (op);

Upon DeliverResult (op, result) do:
$opsToExecute \leftarrow opsToExecute \backslash op$;
  if $|opsToExecute| > 0$ **then**
    Trigger ClientOp(opsToExecute.head);
    Setup Timer OperationTimeout(1000, replicaIpPort, opsToExecute.head);
  end if

Upon Timer OperationTimeout (replica, op) do:

  if $opsToExecute.contains(op)$ **then**
    $replicaIpPort \leftarrow chooseReplica()$;
    Trigger ClientOp(op);
    Setup Timer OperationTimeout(1000, replicaIpPort, op);
  end if

Upon TriggerAskForReplicas() do:
Trigger AskForReplicas(myIpPort);

Upon UpdateReplicas(replicas) do:
$knownReplicas \leftarrow replicas$;

3.2 Application Layer

Algorithm 3: `Aplication Layer`

Interface:
Requests:
ClientOp(op)
OperationResult(op, result)
Indications:
DeliverResult(result)
OrderOperation(op)

State:
askedOperation //list of all the operation requested to this replica
results //map of all the results of the operations performed in the system

Upon Init() do:
$askedOperation \leftarrow \{\}$;
$results \leftarrow \{\}$;

Upon ClientOp(op):
$var \leftarrow results[op.Id]$;
if $op.id \notin results$ **then**
$askedOperation \leftarrow askedOperation \cup op$;
Trigger OrderOperation(op);
else
Trigger DeliverResult(var);
end if

Upon OperationResult (op, result) do:
$results \leftarrow results \cup (op.Id, result)$;
if $op \in askedOperations$ **then**
Trigger DeliverResult(result);
$askedOperations \leftarrow askedOperations \backslash op$;
end if

3.3 State Machine

Algorithm 4: `State Machine (part 1)`

Interface:
Requests:
OrderOperation(op)
Decided(round, op)
LeaderChange(leaderIpPort)
Indications:
Propose (current, op)
OperationResult (op, result)
AddReplica (replicaIpPort)
RemoveReplica (replicaIpPort)

State:
leader //leader ip and port
currentRound //the number of the current round
stateHistory //the actual state machine
nextToExecute //nextOperation to be executed
operationQueue //queue of operations waiting to be decided
dht //the key value store

Upon Init() do:
$leader \leftarrow \perp$; $currentRound \leftarrow 0$;
$stateHistory \leftarrow \{\}$
for each $p \in \pi$
    $stateHistory[p] \leftarrow \{\}$
end for
$nextToExecute \leftarrow 0$
$operationQueue \leftarrow \{\}$
$dht \leftarrow \{\}$
for each $p \in \pi$
    $dht[p] \leftarrow \{\}$
end for

Upon OrderOperation(op) do:
  if $leader == \perp \| (leader != \perp \&\& leader == myIpPort$ **then**
    $operationQueue.enqueue(op)$;
    if $|operationQueue| == 1$ **then**
      Trigger Propose (currentRound, op);
    end if
  else
    Trigger OrderOperation (op);
  end if

Upon LeaderChange (leaderIpPort) do:
$leader \leftarrow leaderIpPort$;

Algorithm 5: `State Machine (part 2)`

```
Decided (round, op) do:
  if stateHistory(round)! = ⊥ then
    op1 ← stateHistory(nextToExecute);
    while op1! = ⊥ do
      opType ← op1.opType
      switch (opType)
      case "read":
        value ← dht[op1.opArg1];
        if value == ⊥ then
          Trigger OperationResult (op1, ⊥);
        else
          Trigger OperationResult (op1, value);
        end if
      case "write":
        olderValue ← dht[op1.opArg1];
        dht[op1.Arg1] ← op1.opArg2;
        if oldValue == ⊥ then
          Trigger OperationResult (op1, ⊥);
        else
          Trigger OperationResult (op1, oldValue);
        end if
      case "add":
        Trigger AddReplica (op1.opArg1);
        Trigger OperationResult (op1, "OK");
      case "remove":
        Trigger AddReplica (op1.opArg1);
        Trigger OperationResult (op1, "OK");
      end switch
      nextToExecute ← nextToExecute + 1;
      opt1 ← stateHistory(nextToExecute)
    end while
    if round == currentRound then
      currentRound ← currentRound + 1;
      if |operationQueue| > 0 then
        if operationQueue.head.opId == op.opId then
          operationQueue.dequeue()
          if |operationQueue| > 0 then
            if leader == ⊥||(leader! = ⊥&&leader == myIpPort then
              Trigger Propose (currentRound, operationQueue.head);
            else
              Trigger OrderOperation (currentRound,
              operationQueue.dequeue());
            end if
          end if
        else
          if leader == ⊥||(leader! = ⊥&&leader == myIpPort then
            Trigger Propose (currentRound, operationQueue.head);
          else
            Trigger OrderOperation (currentRound, operationQueue.dequeue());
          end if
        end if
      end if
    end if
  end if
```

3.4 Multi-Paxos

Algorithm 6: `Multi-Paxos (part 1)`

Interface:
Requests:
Propose (round, operation)
Prepare(replyIpPort, round, number)
PrepareOk(round, number, op)
Accept (round, number, op)
AcceptOk(round, number, op)
AddReplica(replicaIpPort)
RemoveReplica(replicaIpPort)
PingLeader (replicaIpPort)
PingLeaderACK ()
Indications:
LeaderChange(leader)
Decided(currentRound, learnerHiggestacceptvalue)
UpdateReplicas(replicas)

State:
replicas *//set of all replicas*
myIpPort
leader //the leader Ip:port if there is one, null otherwise
currentRound //current known round
proposerSeqNum //Proposer current sequence number
proposerValue //Proposer current value - operation
proposerPrepareOkReceived //PrepareOk received by the proposer
acceptorHighestPrepareSeqNum //Acceptor highest prepare sequence number
acceptorHighestAcceptSeqNum //Acceptor highest accept sequence number
acceptorHighestAcceptValue //Acceptor highest accept value - operation
learnerHighestAcceptSeqNum //Learner highest accept sequence number
learnerHighestAcceptValue //Learner highest accept value - operation
learnerAcceptOkCount //Number of AcceptOk received by the learner
leaderPingCounterb //number of pings sent to leader since last response
leaderPingLimit //limit of pings before assuming the leader died

Upon Init (myPaxosId, myHostName, myPort) do:
$myIpPort \leftarrow myHostname : myPort$;
$replicas \leftarrow \{\}$;
$leader \leftarrow \perp$;
$currentRound \leftarrow 0$;
$proposerSeqNum \leftarrow 0$;
$proposerValue \leftarrow \perp$;
$proposerPrepareOkReceived \leftarrow \{\}$;
$acceptorHighestPrepareSeqNum \leftarrow 0$;
$acceptorHighestAcceptSeqNum \leftarrow 0$;
$acceptorHighestAcceptValue \leftarrow \perp$;
$learnerHighestAcceptSeqNum \leftarrow 0$;
$learnerHighestAcceptValue \leftarrow \perp$;
$learnerAcceptOkCount \leftarrow 0$;
$leaderPingCounter \leftarrow 0$
$leaderPingLimit \leftarrow 3$

Procedure isMajority (count)
return $count \geq (|replicas|/2.0)$;

Procedure isSeqNumGreaterThan (a, b)
return $a > b$;

Algorithm 7: `Multi-Paxos (part 2)`

Procedure isSeqNumGreaterThanOrEqual (a, b):
return $a \geq b$;

Procedure maxValue ():
$sn \leftarrow 0$;
$op \leftarrow \perp$;
foreach $pair \in proposerPrepareOkReceived$ **do:**

if $isSeqNumGreaterThan(pair._1, sn)$ **then**
$sn \leftarrow pair._1$;
$op \leftarrow pair._2$;
end if
return op;

AddReplica (replicaIpPort) do:
$replicas \leftarrow replicas \cup replicaIpPort$

RemoveReplica (replicaIpPort) do:
$replicas \leftarrow replicas \backslash replicaIpPort$

Upon Propose (round, op) do:
if $round \geq currentRound$ **then**
$currentRound \leftarrow round$;
$proposerPrepareOkReceived \leftarrow \{\}$;
if $leader == \perp$ **then**
$proposerSeqNum \leftarrow generateUniqueID()$;
$proposerValue \leftarrow op$;
foreach $replica \in replicas$ **do:**
Trigger Prepare (myIpPort, currentRound, proposerSeqNum);
end for
Setup Timer PrepareMajorityTimeout (5000, currentRound, proposerSeqNum, op);
else if $leader == myIpPort$ **then**
$proposerSeqNum \leftarrow generateUniqueID()$;
$proposerValue \leftarrow op$;
for each $replica \in replicas$ **do:**
Trigger Accept (currentRound, proposerSeqNum, proposerValue);
end for
Setup Timer AcceptMajorityTimeout (5000, currentRound, proposerSeqNum, proposerValue);
end if
end if

Upon Prepare (replyIpPort, round, number) do:

if $round > currentRound$ **then**
$currentRound \leftarrow round$
end if
if $round == currentRound$ **then**
if isSequeNumGreaterThan(number, accpetorHighestPrepareSeqNum)) **then**
$acceptorHighestPrepareSeqNum = number$;
$leader \leftarrow replyIpPort$;
Trigger LeaderChange (leader);
Trigger PrepareOk (currentRound, acceptorHighestAcceptSeqNum, acceptorHighestAcceptValue);
end if
end if

Algorithm 8: `Multi-Paxos (part 3)`

Upon PrepareOk (round, number, op) do:
if $round == currentRound$ **then**
$proposerPrepareOkReceived \leftarrow proposerPrepareOkReceived \cup number$;
if isMajority ($|ProposerPrepareOkReceived|$) **then**
$maxVA \leftarrow maxValue()$;
if $maxVA.opId == \perp$ **then**
$maxVA \leftarrow proposerValue$;
end if
$proposerPrepareOk \leftarrow \{\}$;
for each $replica \in replicas$
Trigger Accept (currentRound, proposerSeqNum, maxVA);
end for
Setup Timer AcceptMajorityTimeout (5000, currentRound, proposerSeqNum, maxVA);
end if
end if

Upon Accept (round, number, op) do:
if $round > currentRound$ **then**
$currentRound \leftarrow round$
end if
if $round == currentRound$ **then**
if isSeqNumGreaterThanorEqual (number, acceptorHighestPrepareSeqNum) **then**
$acceptorHighestAcceptSeqNum \leftarrow number$
$acceptorHighestAcceptValue \leftarrow op$
for each $replica \in replicas$
Trigger AcceptOk (currentRound, acceptorHighestAcceptSeqNum, acceptorHighestAcceptValue);
end for
end if
end if

Upon AcceptOk (round, number, op) do:
if $round > currentRound$ **then**
$currentRound \leftarrow round$
end if
if $round == currentRound$ **then**
if isSeqNumGreaterThanorEqual (number, learnerHighestAcceptSeqNum) **then**
if isSeqNumGreaterThan (number, learnerHighestAcceptSeqNum) **then**
$learnerHighestAcceptSeqNum \leftarrow number$;
$learnerHighestAcceptValue \leftarrow op$;
$learnerAcceptOkCount \leftarrow 0$;
end if
$learnerAcceptOkCount \leftarrow learnerAcceptOkCount + 1$;
if isMajority (learnerAcceptOkCount) **then**
$learnerAcceptOkCount \leftarrow 0$;
Trigger Decided (currentRound, learnerHighestAcceptValue)
end if
end if
end if

Upon PingLeader (replicaIpPort) do:
Trigger PingLeaderACK();

Upon PingLeaderACK (replicaIpPort) do:
$leaderPingCounter \leftarrow 0$;

Algorithm 9: `Multi-Paxos (part 4)`

Upon AskForReplicas(replyIpPort) do:
Trigger UpdateReplicas(replicas);

Upon Periodic Timer TriggerPinLeader () do:

if $leader == \perp$ **then**
if $leaderPingCounter \geq leaderPingLimit$ **then**
$oldLeader \leftarrow leader$;
$leader \leftarrow \perp$;
Trigger LeaderChange (leader);
$leaderPingCount \leftarrow 0$;
$op \leftarrow (myIpPort + " : " + getCurrentTimeStamp() +$
$"\_PROPOSER\_GENERATED", myIpPort, "remove", olderLeader, \perp)$;
Trigger Propose (currentRound, op);
else
Trigger PingLeader (myIpPort);
$leaderPingCount \leftarrow leaderPingCount + 1$;
end if
end if

Upon Timer PrepareMajorityTimeout (round, op) do:
Trigger Propose (round, op);

Upon Timer AcceptMajorityTimeout (round, op) do:
Trigger Propose (round, op);

4 Experimental Evaluation

All of our processes have a command interpreter which allows a user to manually execute certain commands in the system by sending them to the Client layer which will in turn construct and send the operations to the App. The list of the main commands follows:

- `read <key>`: Reads the value associated with the given key
- `write <key> <value>`: Writes the given value for the given key and returns the previous value, if any
- `add <replica>`: Adds a replica to the system
- `remove <replica>`: Removes a replica from the system
- `test <#ops>`: Generate #ops operations to test

Most of our initial evaluation (and our evaluation in general) was based on running the first four commands manually in order to test the various scenarios that might occur during the execution of the program, from the simple read/writes to checking if the system remains functional when the leader process is terminated and the replica membership changes. During this period, our tests, albeit not really scalable, were able to verify the correctness of the program's execution.

Afterwards, we implemented the `test` command, which automatically sends write/read operations (with a distribution of 50% each) to the Client layer in order to simulate a real world application which must be able to handle a big load. All of these operations will eventually go through Multi Paxos in order to maintain the State Machine Replication consistent and once the results are computed and returned to the client, statistics such as the latency of each operation will be calculated.

Finally to further automate this last kind of testing, a shell script named `test.sh` (only tested on git-bash for Windows 10) was created so it can launch the initial set of replicas of the service, as well as a

set of clients which will automatically run the aforementioned `test` command with a specified number of operations.

As for the captured stats, we only captured the timestamp for when each operation is created (StartOnCreation), when it is sent the first time to the app (StartOnSend) and when its result is received (End). This allows us to calculate the On Creation Latency (End - StartOnCreation), On Send Latency (End - StartOnSend), Successful Ops Count, Average on Creation Latency (On Creation Latency / Successful Ops Count), Average on Send Latency (On Send Latency / Successful Ops Count), Throughput (Successful Ops Count / (On Send Latency / 1000)). The results for the last three, when varying the number of clients and operations, are displayed in Table 1. Some notes regarding these results: the number of decimal places used for the throughput was noticeably increased mid-testing; not many tests were made so there might be outliers in the data.

Clients	Ops	AvgCreationLatency	AvgSendLatency	Difference	Throughput
1	50	3214 ms	113 ms	3101 ms	10 ops/s
1	100	5931 ms	107 ms	5824 ms	10 ops/s
1	1000	43972 ms	63 ms	43909 ms	15 ops/s
5	50	14167 ms	516 ms	13651 ms	2 ops/s
5	100	25652 ms	458 ms	25194 ms	2 ops/s
5	1000	104112 ms	154 ms	103958 ms	6 ops/s
10	50	37314 ms	1285 ms	36029 ms	0.777 ops/s
10	100	56369 ms	871 ms	55498 ms	1.146 ops/s
10	1000	224896 ms	431 ms	224465 ms	2.317 ops/s
20	50	44322 ms	1323 ms	42999 ms	0.755 ops/s
20	100	89471 ms	1343 ms	88128 ms	0.744 ops/s
20	1000	317672 ms	568 ms	317104 ms	1.757 ops/s

Table 1: Test results

5 Conclusions

With this project we were able to understand the logic behind Multi-Paxos more in depth and understand it's true complexity. While this project was a little bit easier than the first one it was still hard to do because of our lack of experience with the Akka framework and the Scala language. Regardless, we believe we were able to successfully complete the assignment.

As for our results, it comes as no surprise that the throughput decreases inversely proportional with the number of clients, as the number of competing operations for each round increases. Also, the ever increasing difference between AvgCreationLatency and AvgSendLatency is understandable, as the former takes into account the period of time when an operation is waiting to be executed in the client queue, being an interesting statistic to get the average waiting time until an operation actually is sent to be executed. However, it was a surprise to see it increase with the number of operations which appears to be counter-intuitive and since we can't exactly rule out possible errors in capturing the data or the existence of outliers, we can't conclude much about the probable causes for this behaviour.

References

[1] Alexander, A., *Scala Cookbook* O'Reilly, 2013.

[2] Du, H.; Hilaire, D. J. St., *Multi-Paxos: An Implementation and Evaluation* Dept. of Computer Science and Engineering, Univ. of Washington.

[3] Seif Haridi *Youtube: Lecture 11. Unit 2 From Paxos to Multi-Paxos*. Url: *https://www.youtube.com/watch?v=-Bl5GleEN5s* (accessed: 12-12-2018).