A fail-noisy distributed key-value store implementation

Robin Andersson, Marcus Skagerberg

KTH Information and Communication Technology

Distributed Systems, Advanced Course
Stockholm, Sweden 2016

Royal Institute of Technology
ID2203 Project Report
robin2@kth.se
skag@kth.se

Abstract

The ability to store and handle data in highly available, scalable, responsive manner is a common requirement for today's web services and cloud computing applications. A common solution for providing this is to have a decentralized distributed key-value store.

In this project report, it is described how to implement such a decentralized distributed key-value store with linearizable operation semantics in the fail noisy model using a programming model for distributed systems called Kompics. Processes in the key-value store require consensus to agree on a common value out of values they initially propose to be able to support GET, PUT and CAS-operation.

The implementation features a Multi Paxos algorithm that enables processes to reach consensus and a Replicated State Machine is used to keep all processes within a replication group in consistent states. Test scenarios are presented and executed in order to verify that the implementation satisfies all the abstractions properties.

The key-value store is fault tolerant up to (N-1)/2 failing nodes within a replication group. Availability can be further increased by extending this solution with reconfiguration support for the replication groups and the routing protocol. This would deal with both nodes leaving the system and new nodes joining the system.

Contents

1 Introduction

“A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable.” Failure detectors play an essential role in fault-tolerant distributed systems making them resilient to failing nodes. They provide information about which processes have crashed and which are correct. Though they are not always accurate. A distributed key-value store needs failure detectors in order to be able to function even if some nodes are failing.

In this project report an implementation of a fail noisy distributed key-value store is presented. The implementation handles GET, PUT and CAS operations. In order for the system to be able to handle CAS operations a consensus protocol is needed. Therefore the implementation consists of the following abstractions: Abortable Sequential Consensus, Replicated State machine and Monarchical Eventual Leader Detection. These abstractions requires an Eventually Perfect Failure Detection abstraction and a Best-Effort Broadcast abstraction which both require the Perfect Point To Point Link abstraction.

Implementations are made with Kompics. Kompics is a programming model for distributed systems that implements protocols as event-driven components connected by channels. Kompics is continuously developed by the SICS department at KTH, Royal Institute of Technology.

2 Problem statement

The purpose of the project is to implement and test a simple partitioned and distributed key-value store with linearizable operation semantics.

2.1 Goals

The following goals define the requirements of the implementation.

- The system realize a key-space that is partitioned on different nodes.
- The system assign nodes to different partitions.
- The nodes in the system is able to to detect failure of other nodes.
- The system should provide the functionality to query using a GET operation, for a specific key.
- The system should provide the functionality to store key-value pairs using PUT operations.
- The system should provide the functionality to perform CAS (compare-and-swap) operations.
- The system should have linearizable consistency for GET, PUT and CAS operations.
- The partitioned key-space in the system should be replicated with a specific replication degree δ.

3 Background

3.1 The Fail-noisy abstraction

The Fail-noisy model [1] is a combination of abstractions that expresses assumptions on a distributed system.

3.1.1 Partially synchronous

Partially synchronous [2] is a property that describes the timing assumptions of events in the system. The property entail that the system is asynchronous but eventually becomes synchronous long enough for us to run and terminate algorithms.

3.1.2 Crash-stop process model

The crash-stop process model [3] is an assumption of how to handle node failures. The crash-stop process model entails that nodes does not have access to persistent storage. It means that if a node is restarted then it is consider as an entirely new node in the system as it keeps no state. A node that never crashes is also considered correct.

3.1.3 Perfect links

Perfect links [4] is an abstraction that describes what properties are guaranteed when it comes to sending and delivering messages between processes. Perfect links guarantees that messages are always delivered if the sender and receiver are correct, each message is at most delivered once and that no message is delivered unless it was sent.

3.1.4 Eventually perfect failure detection

Eventually perfect failure detection [5] is an abstraction that is used to detect if a node has crashed. The failure detection is only eventually perfect, which means that there it may suspect nodes to have crashed although they have not. It is however eventually perfect and will after some unknown time only suspect crashed nodes. The detection relies on heartbeats, replies and timeouts to determine if a node has crashed or not.

3.2 Linearizable operation semantics

Linearizability [8] is a property that defines guarantees on a single register on which operations can be invoked, for example read and write. The register could potentially be distributed on several processes. If a register is linearizable then it appears as if when an operation completes, the effect of the operation happens instantaneously in real-time on all the distributed registers. Which means that subsequent reads after a write operation has completed will return the value written.

In order to decide if an execution is linearizable, the concept of linearization points is introduced. A linearization point is a point between the start and end of an operation where the operation takes effect, for example when a value is written or read. If linearization points can be placed in a given execution so that the result of the execution is the same, then it can be considered a linearizable execution. Figure 1 shows an execution with linearization points marked using horizontal arrows. R(6) means that the operation read returned 6 and W(5) means that the operation is a write that writes the value 5.

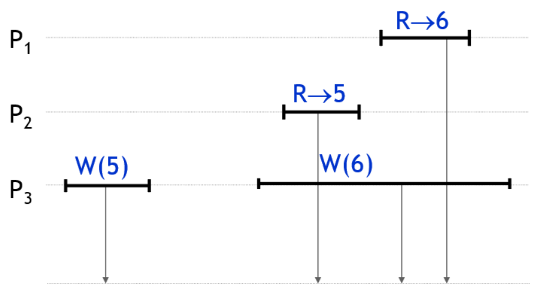


Figure 1: An execution that is linearizable

In figure 1 it is indeed possible to place linearization points to satisfy linearizability. But, if the two read operations were swapped, so that the first one returns 6 and the second one returns 5 it would not be linearizable. There is no way to place linearization points. Figure 2 illustrates this execution.

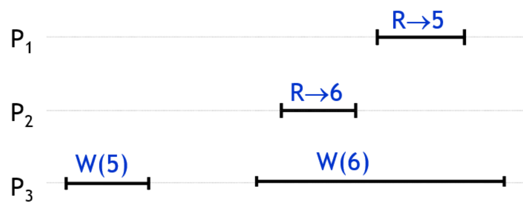


Figure 2: An execution that is not linearizable

3.3 A Distributed Key-Value Store

A key-value store [9] is a data storage paradigm that is used to store and manipulate key-value pairs. A key-value pair consists of a key and an associated value. Common operations on the store is write, read and delete operations. The store is typically distributed over several different nodes.

3.3.1 Key range partitioning

The nodes in the store is assigned to groups where each group is responsible for storing key-value pairs where the keys are in a certain range. The idea is that by separating the key space of the store, the load will be somewhat evenly distributed on the different groups, providing better performance.

3.3.2 Replication groups

Is a given key-value pair stored on only one node or is there one on each node in the group? It would be sufficient to only store one instance of a key-value pair on one node. The drawback of having one instance is that if a node crashes then the data is lost. Therefore there is a need to store multiple instances of a given key-value pair.

Storing multiple instances is referred to as replication, which simply means that there are several nodes, referred to as replicas, that hold the same key-value pair. The degree to which data is replicated is referred to as replication degree. If n nodes in a replication group replicates all key-value pairs in that partition, then the replication degree is n.

Replication is one of the most important properties in a key-value store because it allows for better fault-tolerance and performance in certain cases, however it brings up the problem of data consistency.

3.3.3 Consistency in a key-value store

By allowing replication of values, the problem of keeping all the replicas in a consistent state when key-value pairs are modified is introduced. The need for consistency in the store may be different depending on the application. There are different consistency models that can be applied to the store and one of the strongest is linearizable consistency. Linearizable consistency could be described as creating a linear order of operations that are executed in the system, so that for example reads are guaranteed to return the last value written. Linearizable consistency is achieved by having linearizable operations which is described in section 3.2.

3.3.4 Paxos to ensure linearizable consistency

In order to decide the order of operations to be executed in the system such that the store guarantees linearizable consistency, there is a need for agreement on sequences between all replicas.

Paxos [10] is a family of protocols that solves the problem of agreeing on values. The protocol defines three different roles, proposers that will propose values, acceptors that will accept proposed values and learners that learn the accepted sequence. If Paxos would be implemented in a key-value store, then each node would take on the role of a proposer, acceptor and learner.

In our case the proposed values could be operations. Then it would mean that the sequence of operations is, after having reached agreement, a total order. The operations can therefore be executed in a linear order, that is the same on each replica, ensuring linearizability.

3.3.5 Paxos in the fail-noisy model

Assuming a fail noisy model, it is possible to implement the Paxos protocol. The algorithm does however rely on eventual leader election [11] to ensure that the algorithm does not end up in a non-terminating state, described as the FLP impossiblity. [12]

3.4 The Kompics programming model

3.4.1 Semantics

Components, events and channels are three core concepts of the Kompics programming model [17]. In the model, protocols are implementing by connecting different event-driven components together by channels. Components can communicate between one another by sending different types of events over the channels as Kompics provides a form of type system for events. Every component declares its required and provided ports along with what direction a type of event can be sent in.

3.4.2 Components

Kompics [17] guarantees that components are scheduled on one thread which gives them exclusive access to their internal states. Different components are scheduled in parallel to exploit parallelism. Kompics provides two components that can be used by any application, this is a Timer component that can give timeouts, and Network component that can be used to send data over the network.

3.4.3 Events

Events in kompics [17] are not directly coupled to components, as in Erlang or Akka, instead they are broadcasted across all connected channels. The allows many components to receive one event. Each component decides which events they want to handle and what events they want to ignore by subscribing to event-handlers connected to a specified port.

3.4.4 Channels

The channels provide first-in-first-out (FIFO) order delivery and the delivery of events are sent exactly-once (per receiver). [17] When events are received they queue up at the receiving port until the component handles and executes them.

3.4.5 Simulation

Testing distributed systems is notoriously hard. Kompics [17] provides a system for testing components in simulation scenarios which can be helpful in making pre-deployment tests. One can set up a local environment quickly to test multiple or individual components with the possibility of chaining different operations in different orders.

4 Method

4.1 Process model

This section will contain a brief explanation of our process model.

1. Make assumptions/choices on distributed system abstractions
2. Find out how to actualize the proposed implementations in the Kompics programming model
3. Sequentially implement components mentioned below
4. Write simulation scenarios and run the simulation scenarios to verify the linearizable operation semantics
5. Deploying in a live system

4.2 Implementation

Assuming a fail-noisy model for our implementation, the distributed key-value store ended up consisting of many different abstractions. This section presents these abstractions and the final implementation of the store.

4.2.1 Perfect Links

Several components that were implemented rely on a strong point to point message delivery abstraction, the Perfect Point to Point link. In this assignment, the assumption was made that the Perfect Links abstraction is provided by the TCP/IP network protocol.

Perfect Links can be implemented as suggested by B.1 in Appendix B.

4.2.2 Best-Effort Broadcast

The store needs a basic broadcast abstraction that give us the guarantee that if a sender and the set of receivers is correct, all messages sent from the sender is eventually delivered to the receivers. The Best-Effort [13] broadcast abstraction gives us that guarantee. The case when a broadcast abstraction would be needed is for example when a node in a replication group receives an operation on a key-value pair that is not in its replication group and want to broadcast it to the responsible group. A broadcast is necessary since there are no guarantees that an arbitrary node in that group is alive.

Best-Effort Broadcast is implemented as suggested by B.2 in Appendix B.

4.2.3 Abortable Sequence Consensus

Abortable Sequence Consensus [14] is a variant of Multi-Paxos. The algorithm decides on a continuous stream of values, in our case commands, and is an optimization of the standard Multi-Paxos. The Abortable Sequence Consensus assumes that most of the time there is only one stable proposer. By assuming a single proposer, the prepare phase is only needed once, and after the first prepare it takes 1 roundtrip to decide the next command.

Abortable Sequence Consensus is implemented as suggested by B.3 in Appendix B.

4.2.4 Replicated State Machine

The Replicated State Machine [15] is an abstraction that encapsulates the state of the node. Each replication group replicates the same key-value pairs and all pairs combined, for a replica, are considered a state. The commands that is decided by Abortable Sequence Consensus is executed on the Replicated State Machine.

Replicated State Machine is implemented as suggested by B.4 in Appendix B.

4.2.5 Monarchical Eventual Leader Detection

The Monarchical Eventual Leader Detection [16] abstraction is needed to ensure the liveness of the Abortable Sequence Consensus. If the leader of a replication group should crash, then a leader election needs to be initiated to decide the next leader. Another case when leader election is used is when there are conflicting proposers in Abortable Sequence Consensus and they need to decide on a single proposer. Monarchical Eventual Leader Detection assumes a certain rank that all nodes agree on, and selects the next in line as the leader once the current one is suspected. In our implementation the lower the value of the last octet in the IP-address, the higher the rank of a node. 193.192.0.1 is a higher rank than 193.192.0.2.

Monarchical Eventual Leader Detection is implemented as suggested by B.5 in Appendix B.

4.2.6 Eventually Perfect Failure Detection

In the key-value store, there will be a need for nodes to monitor nodes in their replication group. The Monarchical Eventual Leader Detection relies on the Eventually Perfect Failure Detection to monitor which nodes are considered to be alive and which are suspected to have crashed.

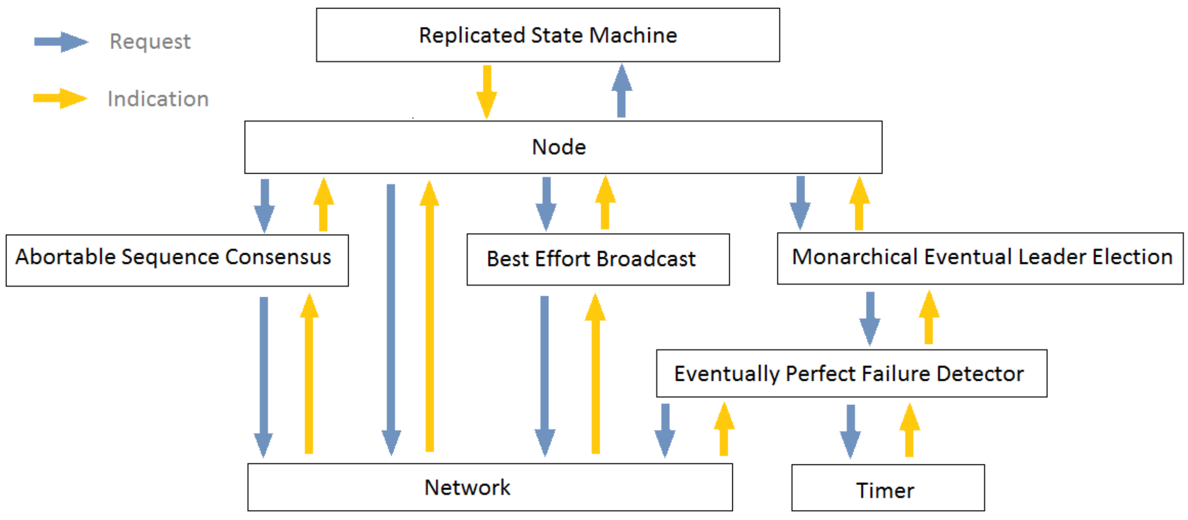
Eventually Perfect Failure Detection is implemented as suggested by B.6 in Appendix B.

4.2.7 Node Infrastructure

The abstractions now need to be connected in a way such that they can interact with each other to implement the desired functionality.

Figure 3 depicts the layering of components on each node in the key-value store. There are two differents flows of interactions which are marked with blue and yellow color. The blue arrow indicates a request, which could for example be to propose a command to the Abortable Sequence Consensus component. The yellow arrow is an indication, which can be thought of as a response to a previous request, for example the next decided command from an Abortable Sequence Consensus component.

The node component serves as the interface to the Replicated State Machine component and wires together Monarchical Eventual Leader Detection component with Abortable Sequence Consensus component.



Request
Indication
Replicated State Machine
Node
Abortable Sequence Consensus
Best Effort Broadcast
Monarchical Eventual Leader Election
Eventually Perfect Failure Detector
Network
Timer

Figure 3: Infrastructural layers

4.2.8 System Infrastructure

Now that the infrastructure of each node is decided, the infrastructure of the store as a whole needs to be decided. Nodes need to be assigned to groups for different key ranges and replication. Also decisions of which process is the initial leader should be made.

Figure 12 proposes a key-value store consisting of three replication groups where each group consists of three nodes. Replication Group 1 will replicate all key-value pairs with keys from 0 to 9, on each node. Replication Group 2 replicates all key-value pairs with keys from 10 to 19. Replication Group 3 replicates all key-value pairs will keys 20 to 29. Since each group consists of three nodes that replicates all key-value pairs in the key range, the replication degree is 3.

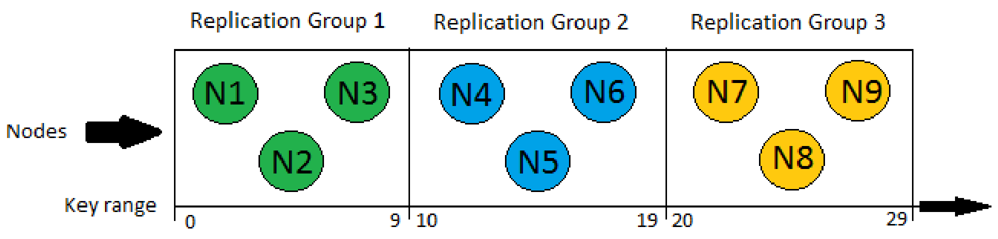


Figure 4: Assigned replication groups

In each replication group, there will be an initial leader. The idea is that the initial leader will be the proposer for Abortable Sequence Consensus in that replication group. However, any node can take the role of a proposer but only the leader in each group is allowed to propose values. Figure 5 depicts the way that the roles for Abortable Sequence Consensus is assigned in the system. N1, N4 and N7 is the leader for their corresponding group.

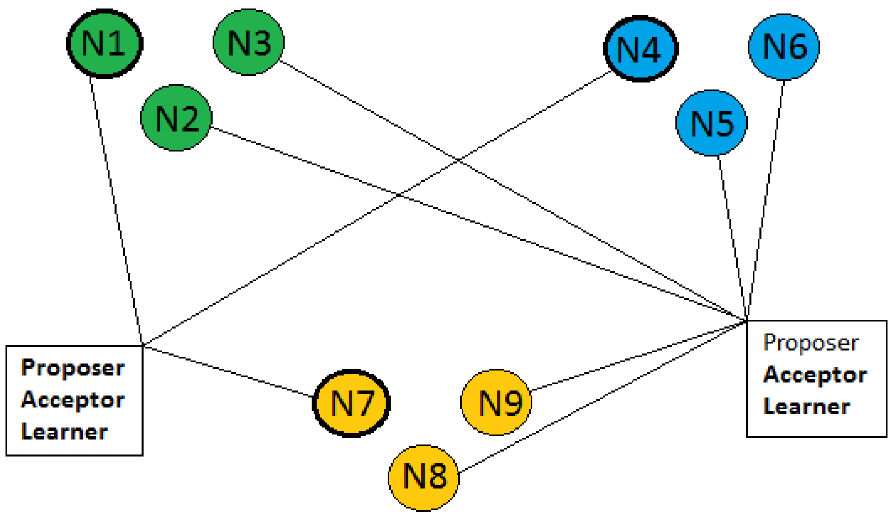


Figure 5: Node roles in the Abortable Sequence Consensus

In a replication group the leader is the only one that can propose a command. Since an external client could contact anyone in the store and still get a response there are some additional routing functionality that needs to be implemented. There are two cases that require forwarding a command to the leader, the first is when a node in the correct replication group receives the command, but is not the leader, in this case the command is simply forwarded to the node is considers the leader. In the case that the node receiving the command is not in the correct replication group, the command is Best-Effort broadcasted to the correct replication group. A broadcast is needed since each node only keeps track of alive nodes in their own replication group.

4.3 Simulation scenarios

In order to check that abstractions properties hold, test scenarios have to be run and verified. It is especially important to verify the linearizable operation semantics that the proposed solution ought to provide. Simulation scenarios are executed for each abstraction within the Fail-noisy model and for other proposed abstrations.

4.3.1 Perfect Point To Point Link

To verify that the Perfect Point To Point Link abstraction, also called Reliable Links abstraction, works as expected the properties PL1, PL2, PL3 [4] have to be satisfied.

- **PL1. Reliable Delivery**: If neither pi nor pj crashes, then every message sent by pi to pj is eventually delivered by pj
- **PL2. No duplication**: Every message is delivered at most once
- **PL3. No creation**: No message is delivered unless it was sent

Scenario

1. Create three replication groups with three nodes in each group
2. A client sends a GET request to leader in the replication group responsible for that key-value pair

System output should indicate that all correct processes deliver, satisfying PL1. That the message is not duplicated, satisfying PL2 And that all delivered messages in the scenario was previously sent by a process. As stated above this implementation assumes that the TCP/IP network protocol satisfies the properties PL1, PL2 and PL3.

4.3.2 Best-Effort Broadcast

To verify that the Best-Effort Broadcast abstraction works as expected the properties BEB1, BEB2 and BEB3 [13] have to be satisfied.

- **BEB1. Validity**: If a correct process broadcasts a message m, then every correct process eventually delivers m
- **BEB2. No duplication**: No message is delivered more than once
- **BEB2. No creation**: If a process delivers a message m with sender s, then m was previously broadcast by process s

Scenario

1. Create three replication groups with three nodes in each group
2. After the nodes have been started a client sends a GET request to a node in a replication group not responsible for that key-value pair.
3. The receiving node X forwards the GET request to the responsible replication group using Best-Effort Broadcast
4. System output shows that each node in the group receives the broadcast according to BEB1, BEB2 and BEB3
5. Kill a node in the responsible replication group
6. A second client sends a GET request which is received by the two correct processes in the responsible replication group

System output should indicate that no message sent by node X should be received no more than one time. If this is true, BEB2 is satisfied. All processes in the best effort broadcast recipient group subscribing on message should receive the message sent by node X. If this is true then it is true that every correct process eventually delivers and BEB1 is satisfied. Only messages sent by node X should be delivered. If this is true, BEB3 is satisfied.

4.3.3 Abortable Sequence Consensus - One leader

To verify that the Abortable Sequence Consensus abstraction properties ASC1, ASC2, ASC3 and ASC4 [14] hold under the assumption that there is one proposer, the following scenario is suggested.

- **ASC1, Validity**: If process p decides v then v is a sequence of proposed commands without duplicates
- **ASC2, Uniform agreement**: If process p decides u and process q decides v then one is a prefix of the other
- **ASC3, Integrity**: If process p decides u and later decides v then u is a prefix of v
- **ASC4, Termination**: If command C is proposed then eventually every correct process decides a sequence containing C

Scenario

1. Create three replication groups with three nodes in each group
2. One client sends a PUT request, another sends a CAS request and a third sends a GET request

System output should indicate that there is only one leader doing PROPOSE, even if requests are sent to different nodes, in a replication group. System output should also indicate that the above mentioned ASC properties are satisfied.

4.3.4 Abortable Sequence Consensus - All leader

To verify that the Abortable Sequence Consensus abstraction properties ASC1, ASC2, ASC3 and ASC4 [14] hold under the assumption that everyone is an active proposer, the following scenario is suggested.

- **ASC1, Validity**: If process p decides v then v is a sequence of proposed commands without duplicates
- **ASC2, Uniform agreement**: If process p decides u and process q decides v then one is a prefix of the other
- **ASC3, Integrity**: If process p decides u and later decides v then u is a prefix of v
- **ASC4, Termination**: If command C is proposed then eventually every correct process decides a sequence containing C

Scenario

1. Create three replication groups with three nodes in each group
2. One client sends a PUT request, another sends a CAS request and a third sends a GET request

System output should indicate that there are multiple leaders doing PROPOSE in a replication group. Two leaders should act as conflicting proposers and ASC still decides on a sequence of commands in total order. System output should also indicate that the above mentioned ASC properties are satisfied.

4.3.5 Abortable Sequence Consensus - No duplicates

To verify that the Abortable Sequence Consensus abstraction properties ASC1, ASC2, ASC3 and ASC4 [14] hold when a proposer PROPOSE the same value multiple times, the following scenario is suggested.

- **ASC1, Validity**: If process p decides v then v is a sequence of proposed commands without duplicates
- **ASC2, Uniform agreement**: If process p decides u and process q decides v then one is a prefix of the other
- **ASC3, Integrity**: If process p decides u and later decides v then u is a prefix of v
- **ASC4, Termination**: If command C is proposed then eventually every correct process decides a sequence containing C

Scenario

1. Create three replication groups with three nodes in each group
2. One client send a GET request, another client send a PUT request and one client send one CAS requests, all processes are started at the same time. Requests from clients are sent to replication groups not responsible for the key-value pairs

System output should indicate that the returned sequence of decided commands, from the ASC, are without duplicates. System output should also indicate that the above mentioned ASC properties are satisfied.

4.3.6 Abortable Sequence Consensus - Leader election

To verify that the Abortable Sequence Consensus abstraction properties ASC1, ASC2, ASC3 and ASC4 [14] hold after a proposer crashes, the following scenario is suggested. Also to verify that the Monarchical Eventual Leader Detection abstraction works with ASC - when a proposer crashes there will be a leader election.

- **ASC1, Validity**: If process p decides v then v is a sequence of proposed commands without duplicates
- **ASC2, Uniform agreement**: If process p decides u and process q decides v then one is a prefix of the other
- **ASC3, Integrity**: If process p decides u and later decides v then u is a prefix of v
- **ASC4, Termination**: If command C is proposed then eventually every correct process decides a sequence containing C

Scenario

1. Create three replication groups with three nodes in each group.
2. Two clients send two GET requests, two clients send two PUT requests and two clients send two CAS requests, all processes are started at the same time.

System output should indicate that leader election works when a leader crashes and that the ASC and ELD properties still holds.

4.3.7 Abortable Sequence Consensus - Quorum majority

To verify that the Abortable Sequence Consensus abstraction properties ASC1, ASC2, ASC3 and ASC4 [14] hold when there is a majority of alive acceptors and does not hold when there is not a majority of alive acceptors, the following scenario is suggested.

- **ASC1, Validity**: If process p decides v then v is a sequence of proposed commands without duplicates
- **ASC2, Uniform agreement**: If process p decides u and process q decides v then one is a prefix of the other
- **ASC3, Integrity**: If process p decides u and later decides v then u is a prefix of v
- **ASC4, Termination**: If command C is proposed then eventually every correct process decides a sequence containing C

Scenario

1. Create three replication groups with three nodes in each group.
2. One client sends a PUT request, another sends a CAS request and a third sends a GET request
3. Two nodes are killed in the targeted replication group
4. Three clients send one request each, same as in step 2

System output should indicate that requests from first round are decided whereas requests from the second round cannot be decided since there is no majority quorum in the responsible replication group. System output should also indicate that the above mentioned ASC properties are satisfied.

4.3.8 Replicated State Machine

To verify that the Replication State Machine abstraction works as expected the properties RSM1 and RSM2 [15] have to be satisfied.

- **RSM1. Agreement**: All correct processes obtain the same sequence of outputs
- **RSM2. Termination**: If a correct process executes a command, then the command eventually produces an output

Scenario

1. Create three replication groups with three nodes in each group
2. Two clients each send a PUT request, two clients send one GET request each and two clients send CAS requests, all processes are started at the same time.

System output should indicate that all processes obtain the same sequence of outputs, satisfying the property RSM1. All executed commands for correct processes produce an output, ensuring that property RSM2 is satisfied. System output also verify linearizability of the operations executed in RSM as the commands are decided in total order which is provided by the ASC abstraction.

4.3.9 Monarchical Eventual Leader Detection

To verify that the Monarchical Eventual Leader Detection abstraction works as expected the properties ELD1 and ELD2 [16] have to be satisfied.

- **ELD1. Eventual accuracy**: There is a time after which every correct process trusts some correct process
- **ELD2. Eventual agreement**: There is a time after which no two correct processes trust different correct processes

Scenario

1. Create three replication groups with three nodes in each group
2. Kill the node with the last octet of 1. (N1)
3. Kill the node with the last octet of 2. (N2)
4. Restart the node with last octet of 2
5. Restart the node with last octet of 1

System output should indicate that N1 is initially leader. When it is killed N2 should take over. Then when N2 is killed, N3 should take over as it is N2's successor. N2 is started again and it should resume leadership. And finally N1 is started again and it resumes leadership. It is now verified that "deaths in the royal family are not final" which is the essential idea of a Monarchical Eventual Leader Detection.

ELD1 is satisfied when logs indicate that all alive and correct processes eventually trusts a correct process as a leader. ELD2 is satisfied when logs indicate that all alive and correct processes eventually trusts the same correct process as a leader. As all nodes agrees on one leader both properties hold.

4.3.10 Eventual Perfect Failure Detection

To verify that Eventual Perfect Failure Detection abstraction works as expected the properties EPFD1 and EPFD2 [5] have to be satisfied.

- **EPFD1. Strong Completeness**: Eventually, every node that crashes is permanently detected by every correct node
- **EPFD2 Eventual Strong Accurary**. Eventually, no correct node is suspected by any correct node

Scenario

1. Create three replication groups with three nodes in each group.
2. Kill the node with the last octet of 1
3. Kill the node with the last octet of 2

System output should indicate that as soon as a process is killed it is suspected by other correct nodes which means that EPFD1 is satisfied. By seeing that all correct nodes are removed from all correct nodes suspect sets, one can verify that EPFD2 is satisfied.

5 Results

5.1 Verification of Simulation Scenarios

The proposed scenarios in section 4.3 are verified by execution in the implemented Key-Value store.

The addressing for the scenario assume:

- Replication group for keys 0-9 on address range 193.192.0.1 - 193.192.0.3
- Replication group for keys 10-19 on address range 193.192.0.4 - 193.192.0.6
- Replication group for keys 20-29 on address range 193.192.0.7 - 193.192.0.9
- 193.192.0.1, 193.192.0.4 and 193.192.0.7 are addresses for the leaders
- All other addresses are external clients calling the store

5.1.1 Perfect Point To Point Link

The Perfect Point To Point Link scenario was a test for the properties of that abstraction. A Client sends a GET request the leader of the replication group that is responsible for the key. Based on the output, the scenario satisfies the properties and the Perfect Point To Point Link works as expected.

See section A.1 in Appendix A for the scenario output.

5.1.2 Best-Effort Broadcast

The Best-Effort Broadcast scenario was a test for the properties of that abstraction. A Client sends a GET request to a replication group that is not responsible for the key. The Best-Effort Broadcast is used to forward the request to all nodes in the responsible replication group. Based on the output, the scenario satisfies the properties since the request is correctly forwarded and the Best-Effort Broadcast works as expected.

See section A.2 in Appendix A for the scenario output.

5.1.3 Abortable Sequence Consensus - One leader

Abortable Sequence Consensus - One leader scenario was a test for the properties of Abortable Sequence Consensus, under the condition that there is only one proposer. Clients send GET, PUT and CAS request to a replication group that is responsible for the key. Based on the output, each replica decides on the same sequence of commands and the abstraction properties are satisfied.

See section A.3 in Appendix A for the scenario output.

5.1.4 Abortable Sequence Consensus - All leader

Abortable Sequence Consensus - All leader scenario was a test for the properties of Abortable Sequence Consensus, under the condition that everyone acts as proposer. Clients send GET, PUT and CAS request to a replication group that is responsible for the key, each node receives the request and proposes the command. Based on the output, each replica decides on the same sequence of commands and there are conflicting proposals, however the abstraction properties are still satisfied.

See section A.4 in Appendix A for the scenario output.

5.1.5 Abortable Sequence Consensus - No duplicates

Abortable Sequence Consensus - No duplicates scenario was a test for the properties of Abortable Sequence Consensus, and specifically that there are no duplicates in the decided sequence. Clients send GET, PUT and CAS request to replication group that is not responsible for the key. This results in a broadcast to the responsible group that receives three command for each initial GET/PUT/-CAS request. So each command is proposed three times. Based on the output, each replica decides on the same sequence of commands and the abstraction properties are satisfied.

See section A.5 in Appendix A for the scenario output.

5.1.6 Abortable Sequence Consensus - Leader election

Abortable Sequence Consensus - Leader election scenario was a test for the properties of Abortable Sequence Consensus in the scenario that the single proposer is killed, and a leader election is needed. In the scenario clients send GET, PUT and CAS request to the replication group responsible for the key. Before the commands are proposed the single proposer is killed. The other nodes in the replication group initiates a new leader election and selects the next node in line. After the leader election is completed more clients send in GET/PUT/-CAS requests, all of which are proposed by the new leader and subsequently decided.

See section A.6 in Appendix A for the scenario output.

5.1.7 Abortable Sequence Consensus - Quorum majority

Abortable Sequence Consensus - Quorum majority scenario was a test for the properties of Abortable Sequence Consensus in the scenario that the majority of acceptors in a replication group is dead. In the scenario clients send GET, PUT and CAS request to the replication group responsible for the key while a majority of acceptors is alive. All nodes in the replication group decides on the same sequence. Then 2 acceptors are killed which means there is only 1 of 3 acceptors alive. The majority of acceptors are dead and while more commands are proposed they are never decided. This is because there is no majority of acceptors choosing the proposal.

See section A.7 in Appendix A for the scenario output.

5.1.8 Replicated State Machine

Replicated State Machine scenario was a test for the properties of that abstraction. In the scenario clients send GET, PUT and CAS request to the replication group responsible for the key. The sequence of commands is decided and each Replicated State Machine executes the commands on their local store. The sequence of commands is correctly executed and response is returned according to the abstraction properties.

See section A.8 in Appendix A for the scenario output.

5.1.9 Monarchical Eventual Leader Detection

Monarchical Eventual Leader Detection scenario was a test for the properties of that abstraction. In the scenario, nodes in a replication group are killed, this triggers a leader election. The leader election satisfies the abstraction properties.

See section A.9 in Appendix A for the scenario output.

5.1.10 Eventual Perfect Failure Detection

Eventual Perfect Failure Detection scenario was a test for the properties of that abstraction. In the scenario, nodes in a replication group are killed. The failure detector correctly suspects crashed nodes and does not suspect alive nodes. The scenario satisfies the abstraction properties.

See section A.10 in Appendix A for the scenario output.

6 Discussion

6.1 Reliability of tests

In previous section the properties of the proposed abstractions were verified. Though these scenarios only show one execution of events received in a non-deterministic sequence. To be completely sure of a system's correctness one would have to test all possible executions or model all possible states which is not realistic. It is assumed that if the properties for an abstraction hold in the proposed tests, the implementation of the abstraction is considered correct.

6.2 Linearizability

Based on the simulation scenarios in section 5.1, it is clear that the GET, PUT and CAS operations performed on the key-value store follows the linearizable semantics. Abortable Sequence Consensus guarantees a total order of decided commands, which are subsequently executed in that same order on each replica in a replication group. The state of the replicas, no matter the operation, will be the same prior to executing each command.

6.3 Advantages of the implementation

This implementation provides fault tolerance up to (N-1)/2 failing nodes within a replication group, making the key-value store resilient to failing nodes. The Eventually Perfect Failure Detection abstraction is implemented and the system is easily extendable with reconfiguration support.

6.4 Limitations of the implementation

This implementation cannot handle events sent to crashed processes. This means that clients can be left waiting, just like in an asynchronous system. It is only once the system reach a partially synchronous state that consensus can be achieved and operations executed. Abortable Sequence Consensus is eventually synchronous which means that there is a time after which the system is synchronous, though it is not known when or for how long. This could leave the client hanging for response. Therefore clients need to perform requests asynchronously and set their own upper bound for timeouts.

In this implementation there is no reconfiguration support that can deal with nodes leaving the system and new nodes joining the system. Every time a reconfiguration is needed the system needs to be restarted which affects the availability of the system.

7 Conclusion

This project report described an implementation of a fail noisy distributed key-value store. The implementation handles GET, PUT and CAS operations. CAS operations require the implementation to include a consensus protocol. The following abstractions are included in the implementation: Abortable Sequence Consensus, Replicated State Machine, Monarchical Eventual Leader Detection, Eventually Perfect Failure Detection, Perfect Point To Point Link and Best-Effort Broadcast.

The system was tested, as earlier described, and successfully deployed.

The store provides linearizable consistency for the supported operations, this is guaranteed by the Abortable Sequence Consensus abstraction and Replicated State Machine.

The key-value store is fault tolerant up to (N-1)/2 failing nodes within a replication group. Availability can be further increased by extending this solution with reconfiguration support for the replication groups and the routing protocol. This would deal with both nodes leaving the system and new nodes joining the system.

8 Contributions

The the project was done pair-programming with two alternating roles, driver and navigator. Robin spend more hours on debugging than Marcus. The project report was divided fairly. Though Robin did put in more hours into the report.

Component , driver , navigator

1. EPFD , Robin, Marcus
2. BEB , Marcus, Robin
3. RIWM , Marcus, Robin
4. MELD , Robin, Marcus
5. ASC , Marcus, Robin
6. RSM , Robin Marcus

9 Bibliography

References

[1] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, p.63.

[2] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, p.47.

[3] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, pp.24-25.

[4] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, pp.37-38.

[5] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, pp.53-56.

[6] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, p.75.

[7] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, p.75.

[8] Herlihy, M. and Wing, J. (1990). Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3), pp.463-492.

[9] Wiese, L. (n.d.). Advanced data management. Berlin: Walter de Gruyter GmbH, p.105.

[10] Lamport, L. (2001). Paxos Made Simple.

[11] Herlihy, M. and Wing, J. (1990). Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3), pp.56-60.

[12] Fischer, M., Lynch, N. and Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2), pp.374-382.

[13] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, pp.75-76.

[14] Haridi, S. and Ekström, N. (2016). Sequence Consensus and Multi-Paxos.

[15] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, pp.330-331.

[16] Cachin, C., Guerraoui, R. and Rodrigues, L. (2011). Introduction to reliable and secure distributed programming. 2nd ed. Berlin: Springer, p.56.

[17] "What Is Kompics? — Kompics 0.9.1 Documentation". Kompics.sics.se. N.p., 2016. Web. 7 Mar. 2016.

Appendix A
System Output from Simulation Scenario

A.1 Perfect Point To Point Link

```
CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.1:10001

/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.1:10001 Decided Sequence so far:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]
/192.193.0.2:10002 Decided Sequence so far:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]
/192.193.0.3:10003 Decided Sequence so far:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

CLIENT/192.193.0.30:10030: Received
    GETREPLY key-5 value-3532 from /192.193.0.1:10001
```

A.2 Best-Effort Broadcast

```
CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.8:10008

/192.193.0.8:10008 Not my key space! Forwarding to correct group: group 1

/192.193.0.8:10008 Sending Best-effort Broadcast
    {GETRequest: pid= 8, seqNum= 0, key= 5}
        to /192.193.0.1:10001
/192.193.0.8:10008 Sending Best-effort Broadcast
    {GETRequest: pid= 8, seqNum= 0, key= 5}
        to /192.193.0.2:10002
```

```
/192.193.0.8:10008 Sending Best−effort Broadcast
    {GETRequest: pid= 8, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.1:10001 Received Best−effort Deliver
    {GETRequest: pid= 8, seqNum= 0, key= 5}
        from /192.193.0.8:10008
/192.193.0.2:10002 Received Best−effort Deliver
    {GETRequest: pid= 8, seqNum= 0, key= 5}
        from /192.193.0.8:10008
/192.193.0.3:10003 Received Best−effort Deliver
    {GETRequest: pid= 8, seqNum= 0, key= 5}
        from /192.193.0.8:10008

/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 8, seqNum= 0, key= 5}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 8, seqNum= 0, key= 5}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 8, seqNum= 0, key= 5}]

/192.193.0.1:10001 Decided Sequence so far:
    [{GETRequest: pid= 8, seqNum= 0, key= 5}]
/192.193.0.2:10002 Decided Sequence so far:
    [{GETRequest: pid= 8, seqNum= 0, key= 5}]
/192.193.0.3:10003 Decided Sequence so far:
    [{GETRequest: pid= 8, seqNum= 0, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 8, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 8, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 8, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 8, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 8, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 8, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 8, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.30:10030

CLIENT/192.193.0.30:10030:
    Received GETREPLY key−5 value−3532
        from /192.193.0.1:10001

/192.193.0.1:10001: Suspect: /192.193.0.2:10002
/192.193.0.3:10003: Suspect: /192.193.0.2:10002

CLIENT/192.193.0.31:10031 Sent
    {GETRequest: pid= 31, seqNum= 0, key= 5}
        to /192.193.0.8:10008

/192.193.0.8:10008 Not my key space!
```

```
    Forwarding to correct group: group 1

/192.193.0.8:10008 Sending Best-effort Broadcast
    {GETRequest: pid= 8, seqNum= 1, key= 5}
        to /192.193.0.1:10001
/192.193.0.8:10008 Sending Best-effort Broadcast
    {GETRequest: pid= 8, seqNum= 1, key= 5}
        to /192.193.0.2:10002
/192.193.0.8:10008 Sending Best-effort Broadcast
    {GETRequest: pid= 8, seqNum= 1, key= 5}

        to /192.193.0.3:10003
/192.193.0.1:10001 Received Best-effort Deliver
    {GETRequest: pid= 8, seqNum= 1, key= 5}
        from /192.193.0.8:10008
/192.193.0.3:10003 Received Best-effort Deliver
    {GETRequest: pid= 8, seqNum= 1, key= 5}
        from /192.193.0.8:10008

/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 8, seqNum= 1, key= 5}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 8, seqNum= 1, key= 5}]

/192.193.0.1:10001 Decided Sequence so far:
    [{GETRequest: pid= 8, seqNum= 0, key= 5},
        {GETRequest: pid= 8, seqNum= 1, key= 5}]
/192.193.0.3:10003 Decided Sequence so far:
    [{GETRequest: pid= 8, seqNum= 0, key= 5},
        {GETRequest: pid= 8, seqNum= 1, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 8, seqNum= 1, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 8, seqNum= 1, key= 5, value= 3532, successful= true}
        to /192.193.0.31:10031

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 8, seqNum= 1, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 8, seqNum= 1, key= 5, value= 3532, successful= true}
        to /192.193.0.31:10031

/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 8, seqNum= 1, key= 5, value= 3532, successful= true}
        to /192.193.0.31:10031

CLIENT/192.193.0.31:10031: Received
    GETREPLY key-5 value-3532
        from /192.193.0.1:10001
```

A.3 Abortable Sequence Consensus - One leader

```
CLIENT/192.193.0.10:10010 Sent
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001

CLIENT/192.193.0.11:10011
    Sent {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.2:10002
```

```
CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]

/192.193.0.1:10001 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.1:10001 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.2:10002 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.2:10002 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.3:10003 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.3:10003 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.1:10001 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.1:10001 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.2:10002 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.2:10002 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.3:10003 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.3:10003 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030
/192.193.0.1:10001 Leader Node Sends Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.1:10001 Leader Node Sends Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

CLIENT/192.193.0.10:10010:
    Received PUTREPLY: key-5 value-1 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.11:10011: Received
    CASREPLY: key-5 referenceValue-1 newValue-30 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.30:10030:
    Received GETREPLY key-5 value-30
        from /192.193.0.1:10001
```

A.4 Abortable Sequence Consensus - All leader

```
CLIENT/192.193.0.10:10010 Sent
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001
```

```
CLIENT/192.193.0.11:10011 Sent
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.2:10002

CLIENT/192.193.0.12:10012 Sent
    {GETRequest: pid= 12, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.2:10002 Proposing:
    [{CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]
/192.193.0.3:10003 Proposing:
    [{GETRequest: pid= 12, seqNum= 0, key= 5}]

/192.193.0.1:10001 Received AscAbort, checking for new trust...
/192.193.0.2:10002 Received AscAbort, checking for new trust...

/192.193.0.1:10001 Received Trust, new leader is /192.193.0.1:10001
/192.193.0.1:10001 Forwarding HBQ-MSG
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001
/192.193.0.2:10002 Received Trust, new leader is /192.193.0.1:10001
/192.193.0.2:10002 Forwarding HBQ-MSG
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.1:10001

/192.193.0.1:10001 Proposing: [{PUTRequest:
    pid= 10, seqNum= 0, key= 5, value= 1}]

/192.193.0.2:10002 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5}]

/192.193.0.3:10003 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5}]

/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 12, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 12, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.12:10012

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 12, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 12, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.12:10012

/192.193.0.1:10001 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5}]

/192.193.0.3:10003 Leader Node Sends Response:
    {GETReply: pid= 12, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.12:10012

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 12, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 12, seqNum= 0, key= 5, value= 3532, successful= true}
```

```
        to /192.193.0.12:10012
/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 12, seqNum= 0, key= 5, value= 3532, successful= true}
        to /192.193.0.12:10012

CLIENT/192.193.0.12:10012:
    Received GETREPLY key-5 value-3532
        from /192.193.0.3:10003

CLIENT/192.193.0.12:10012: Received
    GETREPLY key-5 value-3532
        from /192.193.0.1:10001

/192.193.0.1:10001 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5},
        {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.2:10002 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5},
        {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.3:10003 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5},
        {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]

/192.193.0.1:10001 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.1:10001 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.1:10001 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5},
        {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
            {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.2:10002 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.2:10002 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.2:10002 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5},
        {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
            {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.3:10003 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.3:10003 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.3:10003 Decided Sequence so far:
    [{GETRequest: pid= 12, seqNum= 0, key= 5},
        {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
            {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.1:10001 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.1:10001 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.1:10001 Leader Node Sends Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
```

```
/192.193.0.2:10002 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.2:10002 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

/192.193.0.3:10003 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.3:10003 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.3:10003 Leader Node Sends Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.1:10001 Leader Node Sends Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.3:10003 Leader Node Sends Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

CLIENT/192.193.0.10:10010: Received
    PUTREPLY: key-5 value-1 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.11:10011: Received
    CASREPLY: key-5 referenceValue-1 newValue-30 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.10:10010: Received
    PUTREPLY: key-5 value-1 success-true
        from /192.193.0.3:10003

CLIENT/192.193.0.11:10011:
    Received CASREPLY: key-5 referenceValue-1 newValue-30 success-true
        from /192.193.0.3:10003
```

A.5 Abortable Sequence Consensus - No duplicates

```
CLIENT/192.193.0.10:10010 Sent
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.5:10005

CLIENT/192.193.0.15:10015 Sent
    {CASRequest: pid= 15, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.9:10009

CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.7:10007

/192.193.0.5:10005 Not my key space! Forwarding to correct group: group 1
/192.193.0.9:10009 Not my key space! Forwarding to correct group: group 1
/192.193.0.7:10007 Not my key space! Forwarding to correct group: group 1

/192.193.0.5:10005 Sending Best-effort Broadcast
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001
/192.193.0.5:10005 Sending Best-effort Broadcast
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
```

```
        to /192.193.0.2:10002
/192.193.0.5:10005 Sending Best−effort Broadcast
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
        to /192.193.0.3:10003
/192.193.0.9:10009 Sending Best−effort Broadcast
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.1:10001
/192.193.0.9:10009 Sending Best−effort Broadcast
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.2:10002
/192.193.0.9:10009 Sending Best−effort Broadcast
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.3:10003
/192.193.0.7:10007 Sending Best−effort Broadcast
    {GETRequest: pid= 7, seqNum= 0, key= 5}
        to /192.193.0.1:10001
/192.193.0.7:10007 Sending Best−effort Broadcast
    {GETRequest: pid= 7, seqNum= 0, key= 5}
        to /192.193.0.2:10002
/192.193.0.7:10007 Sending Best−effort Broadcast
    {GETRequest: pid= 7, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.1:10001 Received Best−effort Deliver
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
        from /192.193.0.5:10005
/192.193.0.2:10002 Received Best−effort Deliver
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
        from /192.193.0.5:10005
/192.193.0.3:10003 Received Best−effort Deliver
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
        from /192.193.0.5:10005
/192.193.0.1:10001 Received Best−effort Deliver
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
        from /192.193.0.9:10009
/192.193.0.2:10002 Received Best−effort Deliver
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
        from /192.193.0.9:10009
/192.193.0.3:10003 Received Best−effort Deliver
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
        from /192.193.0.9:10009
/192.193.0.1:10001 Received Best−effort Deliver
    {GETRequest: pid= 7, seqNum= 0, key= 5}
        from /192.193.0.7:10007
/192.193.0.2:10002 Received Best−effort Deliver
    {GETRequest: pid= 7, seqNum= 0, key= 5}
        from /192.193.0.7:10007
/192.193.0.3:10003 Received Best−effort Deliver
    {GETRequest: pid= 7, seqNum= 0, key= 5}
        from /192.193.0.7:10007

/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}]
/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}]
/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}]
/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}]
/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}]
/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 7, seqNum= 0, key= 5}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 7, seqNum= 0, key= 5}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 7, seqNum= 0, key= 5}]

/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}]
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}]
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}]

/192.193.0.1:10001 RSM Executing:
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
/192.193.0.1:10001 RSM Response:
    {PUTReply: pid= 5, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.2:10002 RSM Executing:
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
/192.193.0.2:10002 RSM Response:
    {PUTReply: pid= 5, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.3:10003 RSM Executing:
    {PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1}
/192.193.0.3:10003 RSM Response:
    {PUTReply: pid= 5, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.1:10001 RSM Executing:
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.1:10001 RSM Response:
    {CASReply: pid=9, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.15:10015
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 7, seqNum= 0, key= 5}]

/192.193.0.2:10002 RSM Executing:
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.2:10002 RSM Response:
    {CASReply: pid=9, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.15:10015
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 7, seqNum= 0, key= 5}]
```

```
/192.193.0.3:10003 RSM Executing:
    {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.3:10003 RSM Response:
    {CASReply: pid=9, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.15:10015
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 5, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 9, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 7, seqNum= 0, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 7, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 7, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030
/192.193.0.1:10001 Leader Node Sends Response:
    {PUTReply: pid= 5, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 7, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 7, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 7, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 7, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.1:10001 Leader Node Sends Response:
    {CASReply: pid=9, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.15:10015
/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 7, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

CLIENT/192.193.0.10:10010: Received
    PUTREPLY: key-5 value-1 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.15:10015: Received
    CASREPLY: key-5 referenceValue-1 newValue-30 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.30:10030: Received
    GETREPLY key-5 value-30
        from /192.193.0.1:10001
```

A.6 Abortable Sequence Consensus - Leader election

```
CLIENT/192.193.0.10:10010 Sent
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001

CLIENT/192.193.0.15:10015 Sent
    {CASRequest: pid= 15, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.2:10002
```

```
CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.2:10002: Suspect: /192.193.0.1:10001
/192.193.0.3:10003: Suspect: /192.193.0.1:10001
/192.193.0.2:10002 Received Trust,
    new leader is /192.193.0.2:10002
/192.193.0.3:10003 Received Trust,
    new leader is /192.193.0.2:10002

CLIENT/192.193.0.11:10011 Sent
    {PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2}
        to /192.193.0.2:10002

CLIENT/192.193.0.16:10016 Sent
    {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30}
        to /192.193.0.3:10003

CLIENT/192.193.0.31:10031 Sent
    {GETRequest: pid= 31, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.2:10002 Proposing:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2}]
/192.193.0.2:10002 Proposing:
    [{CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30}]
/192.193.0.2:10002 Proposing:
    [{GETRequest: pid= 31, seqNum= 0, key= 5}]

/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2}]
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2}]

/192.193.0.2:10002 RSM Executing:
    {PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2}
/192.193.0.2:10002 RSM Response:
    {PUTReply: pid= 11, seqNum= 0, key= 5, value= 2, successful= true}
        to /192.193.0.11:10011
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2},
        {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30}]

/192.193.0.3:10003 RSM Executing:
    {PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2}
/192.193.0.3:10003 RSM Response:
    {PUTReply: pid= 11, seqNum= 0, key= 5, value= 2, successful= true}
        to /192.193.0.11:10011
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2},
        {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30}]

/192.193.0.2:10002 RSM Executing:
    {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30}
/192.193.0.2:10002 RSM Response:
    {CASReply: pid=16, seqNum=0, key=5, value=30, oldValue=2, successful=true}
        to /192.193.0.16:10016
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2},
        {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30},
            {GETRequest: pid= 31, seqNum= 0, key= 5}]
```

```
/192.193.0.3:10003 RSM Executing:
    {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30}
/192.193.0.3:10003 RSM Response:
    {CASReply: pid=16, seqNum=0, key=5, value=30, oldValue=2, successful=true}
        to /192.193.0.16:10016
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 11, seqNum= 0, key= 5, value= 2},
        {CASRequest: pid= 16, seqNum= 0, key= 5, value= 2, newValue= 30},
            {GETRequest: pid= 31, seqNum= 0, key= 5}]

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 31, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 31, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.31:10031
/192.193.0.2:10002 Leader Node Sends Response:
    {PUTReply: pid= 11, seqNum= 0, key= 5, value= 2, successful= true}
    to /192.193.0.11:10011

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 31, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 31, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.31:10031

/192.193.0.2:10002 Leader Node Sends Response:
    {CASReply: pid=16, seqNum=0, key=5, value=30, oldValue=2, successful=true}
        to /192.193.0.16:10016

/192.193.0.2:10002 Leader Node Sends Response:
    {GETReply: pid= 31, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.31:10031

CLIENT/192.193.0.11:10011: Received
    PUTREPLY: key-5 value-2 success-true
        from /192.193.0.2:10002

CLIENT/192.193.0.16:10016: Received
    CASREPLY: key-5 referenceValue-2 newValue-30 success-true
        from /192.193.0.2:10002

CLIENT/192.193.0.31:10031: Received
    GETREPLY key-5 value-30
        from /192.193.0.2:10002
```

A.7 Abortable Sequence Consensus - Quorum majority

```
CLIENT/192.193.0.10:10010 Sent
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001

/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]

/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]

/192.193.0.1:10001 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.1:10001 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

/192.193.0.2:10002 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.2:10002 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

/192.193.0.3:10003 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.3:10003 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

/192.193.0.1:10001 Leader Node Sends Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

CLIENT/192.193.0.10:10010: Received
    PUTREPLY: key-5 value-1 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.11:10011 Sent
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.2:10002

/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.1:10001 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.1:10001 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

/192.193.0.2:10002 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.2:10002 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

/192.193.0.3:10003 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.3:10003 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
```

```
        to /192.193.0.11:10011

/192.193.0.1:10001 Leader Node Sends Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

CLIENT/192.193.0.11:10011: Received
    CASREPLY: key-5 referenceValue-1 newValue-30 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.3:10003

/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 30, seqNum= 0, key= 5}]
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30},
            {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 30, successful= true}
        to /192.193.0.30:10030

CLIENT/192.193.0.30:10030: Received
    GETREPLY key-5 value-30 from /192.193.0.1:10001

/192.193.0.1:10001: Suspect: /192.193.0.2:10002
/192.193.0.3:10003: Suspect: /192.193.0.2:10002
/192.193.0.1:10001: Suspect: /192.193.0.3:10003

CLIENT/192.193.0.15:10015 Sent
    {PUTRequest: pid= 15, seqNum= 0, key= 5, value= 1} to /192.193.0.1:10001
```

```
/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 15, seqNum= 0, key= 5, value= 1}]

CLIENT/192.193.0.16:10016 Sent
    {CASRequest: pid= 16, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.2:10002

CLIENT/192.193.0.40:10040 Sent
    {GETRequest: pid= 40, seqNum= 0, key= 5}
        to /192.193.0.3:10003
```

A.8 Replicated State Machine

```
CLIENT/192.193.0.10:10010 Sent
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
        to /192.193.0.1:10001

CLIENT/192.193.0.30:10030 Sent
    {GETRequest: pid= 30, seqNum= 0, key= 5}
        to /192.193.0.1:10001

CLIENT/192.193.0.11:10011 Sent
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
        to /192.193.0.3:10003

/192.193.0.1:10001 Proposing:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.1:10001 Proposing:
    [{GETRequest: pid= 30, seqNum= 0, key= 5}]
/192.193.0.1:10001 Proposing:
    [{CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}]

/192.193.0.1:10001 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.1:10001 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.2:10002 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.2:10002 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.3:10003 RSM Executing:
    {PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1}
/192.193.0.3:10003 RSM Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {GETRequest: pid= 30, seqNum= 0, key= 5}]

/192.193.0.1:10001 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.1:10001 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.30:10030
/192.193.0.1:10001 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {GETRequest: pid= 30, seqNum= 0, key= 5},
            {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.2:10002 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.2:10002 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.30:10030
/192.193.0.2:10002 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {GETRequest: pid= 30, seqNum= 0, key= 5},
            {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.3:10003 RSM Executing:
    {GETRequest: pid= 30, seqNum= 0, key= 5}
/192.193.0.3:10003 RSM Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.30:10030
/192.193.0.3:10003 Decided Sequence so far:
    [{PUTRequest: pid= 10, seqNum= 0, key= 5, value= 1},
        {GETRequest: pid= 30, seqNum= 0, key= 5},
            {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}]

/192.193.0.1:10001 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.1:10001 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011
/192.193.0.1:10001 Leader Node Sends Response:
    {PUTReply: pid= 10, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.10:10010

/192.193.0.2:10002 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.2:10002 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

/192.193.0.3:10003 RSM Executing:
    {CASRequest: pid= 11, seqNum= 0, key= 5, value= 1, newValue= 30}
/192.193.0.3:10003 RSM Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
        to /192.193.0.11:10011

/192.193.0.1:10001 Leader Node Sends Response:
    {GETReply: pid= 30, seqNum= 0, key= 5, value= 1, successful= true}
        to /192.193.0.30:10030
/192.193.0.1:10001 Leader Node Sends Response:
    {CASReply: pid=11, seqNum=0, key=5, value=30, oldValue=1, successful=true}
```

```
        to /192.193.0.11:10011

CLIENT/192.193.0.10:10010: Received
    PUTREPLY: key-5 value-1 success-true
        from /192.193.0.1:10001

CLIENT/192.193.0.30:10030: Received
    GETREPLY key-5 value-1
        from /192.193.0.1:10001

CLIENT/192.193.0.11:10011: Received
    CASREPLY: key-5 referenceValue-1 newValue-30 success-true
        from /192.193.0.1:10001
```

A.9 Monarchical Eventual Leader Detection

```
/192.193.0.2:10002: Suspect: /192.193.0.1:10001
/192.193.0.3:10003: Suspect: /192.193.0.1:10001

/192.193.0.2:10002 Received Trust,
    new leader is /192.193.0.2:10002
/192.193.0.3:10003 Received Trust,
    new leader is /192.193.0.2:10002

/192.193.0.3:10003: Suspect: /192.193.0.2:10002
/192.193.0.3:10003 Received Trust,
    new leader is /192.193.0.3:10003

/192.193.0.2:10002: Starting up

/192.193.0.2:10002: My leader is /192.193.0.1:10001

/192.193.0.3:10003: Restore: /192.193.0.2:10002
/192.193.0.3:10003 Received Trust,
    new leader is /192.193.0.2:10002

/192.193.0.2:10002: Suspect: /192.193.0.1:10001
/192.193.0.2:10002 Received Trust,
    new leader is /192.193.0.2:10002

/192.193.0.1:10001: Starting up
/192.193.0.1:10001: My leader is /192.193.0.1:10001

/192.193.0.2:10002: Restore: /192.193.0.1:10001
/192.193.0.2:10002 Received Trust,
    new leader is /192.193.0.1:10001

/192.193.0.3:10003: Restore: /192.193.0.1:10001
/192.193.0.3:10003 Received Trust,
    new leader is /192.193.0.1:10001
```

A.10 Eventual Perfect Failure Detection

```
/192.193.0.2:10002: Suspect: /192.193.0.1:10001
/192.193.0.3:10003: Suspect: /192.193.0.1:10001

/192.193.0.2:10002 Received Trust,
    new leader is /192.193.0.2:10002
```

```
/192.193.0.3:10003 Received Trust,
   new leader is /192.193.0.2:10002

/192.193.0.3:10003: Suspect: /192.193.0.2:10002
/192.193.0.3:10003 Received Trust,
   new leader is /192.193.0.3:10003
```

Appendix B
Algorithms

B.1 Perfect Links

Algorithm 2.2: Eliminate Duplicates

```
Implements:
    PerfectPointToPointLinks, instance pl.

Uses:
    StubbornPointToPointLinks, instance sl.

upon event ⟨ pl, Init ⟩ do
    delivered := $\emptyset$;

upon event ⟨ pl, Send | $q, m$ ⟩ do
    trigger ⟨ sl, Send | $q, m$ ⟩;

upon event ⟨ sl, Deliver | $p, m$ ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup \{m\}$;
        trigger ⟨ pl, Deliver | $p, m$ ⟩;
```

Figure 6: Perfect Links pseudocode

B.2 Best-Effort Broadcast

Algorithm 3.1: Basic Broadcast

```
Implements:
    BestEffortBroadcast, instance beb.

Uses:
    PerfectPointToPointLinks, instance pl.

upon event ⟨ beb, Broadcast | $m$ ⟩ do
    forall $q \in \Pi$ do
        trigger ⟨ pl, Send | $q, m$ ⟩;

upon event ⟨ pl, Deliver | $p, m$ ⟩ do
    trigger ⟨ beb, Deliver | $p, m$ ⟩;
```

Figure 7: Best-Effort broadcast pseudocode

B.3 Abortable Sequence Consensus

Algorithm 1 Multi-Paxos: Prepare Phase

```
Implements:
    AbortableSequenceConsensus, instance asc.
Uses:
    FIFOPerfectPointToPointLinks, instance fpl.

upon event ⟨ asc, Init ⟩ do
    $t := 0$;                                   ▷ logical clock
    $prepts := 0$;                              ▷ acceptor: prepared timestamp
    $(ats, av, al) := (0, \langle\rangle, 0)$;   ▷ acceptor: timestamp, accepted seq, length of decided seq
    $(pts, pv, pl) := (0, \langle\rangle, 0)$;   ▷ proposer: timestamp, proposed seq, length of learned seq
    $proposedValues := \langle\rangle$;          ▷ proposer: values proposed while preparing
    $readlist := [\bot]^N$;
    $accepted := [0]^N$;     ▷ proposer's knowledge about length of acceptor's longest accepted seq
    $decided := [0]^N$;      ▷ proposer's knowledge about length of acceptor's longest decided seq

upon event ⟨ asc, Propose | v ⟩ do
    $t := t + 1$;
    if $pts = 0$ then
        $pts := t \times N + rank(self)$;
        $pv := prefix(av, al)$;
        $pl := 0$;
        $proposedValues := \langle v \rangle$;
        $readlist := [\bot]^N$;
        $accepted := [0]^N$;
        $decided := [0]^N$;
        for all $p \in \Pi$ do
            trigger ⟨ fpl, Send | p, [PREPARE, pts, al, t] ⟩;
    else if $\#(readlist) \leq \lfloor N/2 \rfloor$ then
        $proposedValues := proposedValues + \langle v \rangle$;   ▷ append to sequence
    else if $v \notin pv$ then
        $pv := pv + \langle v \rangle$;
        for all $p \in \Pi$ such that $readlist[p] \neq \bot$ do
            trigger ⟨ fpl, Send | p, [ACCEPT, pts, ⟨v⟩, #(pv) − 1, t] ⟩;

upon event ⟨ fpl, Deliver | q, [PREPARE, ts, l, t'] ⟩ do
    $t := max(t, t') + 1$;
    if $ts < prepts$ then
        trigger ⟨ fpl, Send | q, [NACK, ts, t] ⟩;
    else
        $prepts := ts$;
        trigger ⟨ fpl, Send | q, [PREPAREACK, ts, ats, suffix(av, l), al, t] ⟩;

upon event ⟨ fpl, Deliver | q, [NACK, pts', t'] ⟩ do
    $t := max(t, t') + 1$;
    if $pts' = pts$ then
        $pts := 0$;
        trigger ⟨ asc, Abort ⟩
```

Figure 8: Abortable Sequence Consensus - Prepare Phase pseudocode

Algorithm 2 Multi-Paxos: Accept Phase

```
upon event $\langle$ fpl, Deliver | q, [PrepareAck, $pts', ts, vsuf, l, t'$] $\rangle$ do
    $t := max(t, t') + 1$;
    if $pts' = pts$ then
        $readlist[q] := (ts, vsuf)$;
        $decided[q] := l$;
        if $\#(readlist) = \lfloor N/2 \rfloor + 1$ then
            $(ts', vsuf') := (0, \langle\rangle)$;
            for all $(ts'', vsuf'') \in readlist$ do
                if $ts' < ts'' \vee (ts' = ts'' \wedge \#(vsuf') < \#(vsuf''))$ then
                    $(ts', vsuf') := (ts'', vsuf'')$;
            $pv := pv + vsuf'$;
            for all $v \in proposedValues$ such that $v \notin pv$ do
                $pv := pv + \langle v \rangle$;
            for all $p \in \Pi$ such that $readlist[p] \neq \bot$ do
                $l' := decided[p]$;
                trigger $\langle$ fpl, Send | p, [Accept, $pts, suffix(pv, l'), l', t$] $\rangle$;
        else if $\#(readlist) > \lfloor N/2 \rfloor + 1$ then
            trigger $\langle$ fpl, Send | q, [Accept, $pts, suffix(pv, l), l, t$] $\rangle$;
            if $pl \neq 0$ then
                trigger $\langle$ fpl, Send | q, [Decide, $pts, pl, t$] $\rangle$;

upon event $\langle$ fpl, Deliver | q, [Accept, $ts, vsuf, offs, t'$] $\rangle$ do
    $t := max(t, t') + 1$;
    if $ts \neq prepts$ then
        trigger $\langle$ fpl, Send | q, [Nack, $ts, t$] $\rangle$;
    else
        $ats := ts$;
        if $offs < \#(av)$ then
            $av := prefix(av, offs)$;                              ▷ truncate sequence
        $av := av + vsuf$;
        trigger $\langle$ fpl, Send | q, [AcceptAck, $ts, \#(av), t$] $\rangle$;

upon event $\langle$ fpl, Deliver | q, [AcceptAck, $pts', l, t'$] $\rangle$ do
    $t := max(t, t') + 1$;
    if $pts' = pts$ then
        $accepted[q] := l$;
        if $pl < l \wedge \#(\{p \in \Pi \mid accepted[p] \geq l\}) > \lfloor N/2 \rfloor$ then
            $pl := l$;
            for all $p \in \Pi$ such that $readlist[p] \neq \bot$ do
                trigger $\langle$ fpl, Send | p, [Decide, $pts, pl, t$] $\rangle$;

upon event $\langle$ fpl, Deliver | q, [Decide, $ts, l, t'$] $\rangle$ do
    $t := max(t, t') + 1$;
    if $ts = prepts$ then
        while $al < l$ do
            trigger $\langle$ asc, Decide | $av[al]$ $\rangle$;          ▷ zero-based indexing
            $al := al + 1$;
```

Figure 9: Abortable Sequence Consensus - Accept Phase pseudocode

B.4 Replicated State Machine

Algorithm 6.13: Replicated State Machine using Total-Order Broadcast

```
Implements:
    ReplicatedStateMachine, instance rsm.

Uses:
    UniformTotalOrderBroadcast, instance utob;

upon event ⟨ rsm, Init ⟩ do
    state := initial state;

upon event ⟨ rsm, Execute | command ⟩ do
    trigger ⟨ utob, Broadcast | command ⟩;

upon event ⟨ utob, Deliver | p, command ⟩ do
    (response, newstate) := execute(command, state);
    state := newstate;
    trigger ⟨ rsm, Output | response ⟩;
```

Figure 10: Replicated State Machine pseudocode

B.5 Monarchical Eventual Leader Detection

Algorithm 2.8: Monarchical Eventual Leader Detection

```
Implements:
    EventualLeaderDetector, instance $\Omega$.

Uses:
    EventuallyPerfectFailureDetector, instance $\Diamond\mathcal{P}$.

upon event ⟨ $\Omega$, Init ⟩ do
    suspected := $\emptyset$;
    leader := $\bot$;

upon event ⟨ $\Diamond\mathcal{P}$, Suspect | p ⟩ do
    suspected := suspected $\cup$ {p};

upon event ⟨ $\Diamond\mathcal{P}$, Restore | p ⟩ do
    suspected := suspected $\setminus$ {p};

upon leader $\neq$ maxrank($\Pi \setminus$ suspected) do
    leader := maxrank($\Pi \setminus$ suspected);
    trigger ⟨ $\Omega$, Trust | leader ⟩;
```

Figure 11: Monarchical Eventual Leader Detection pseudocode

B.6 Eventually Perfect Failure Detection

Algorithm 2.7: Increasing Timeout

```
Implements:
    EventuallyPerfectFailureDetector, instance $\Diamond\mathcal{P}$.

Uses:
    PerfectPointToPointLinks, instance $pl$.

upon event $\langle \Diamond\mathcal{P}, Init \rangle$ do
    $alive := \Pi$;
    $suspected := \emptyset$;
    $delay := \Delta$;
    $starttimer(delay)$;

upon event $\langle Timeout \rangle$ do
    if $alive \cap suspected \neq \emptyset$ then
        $delay := delay + \Delta$;
    forall $p \in \Pi$ do
        if $(p \notin alive) \wedge (p \notin suspected)$ then
            $suspected := suspected \cup \{p\}$;
            trigger $\langle \Diamond\mathcal{P}, Suspect \mid p \rangle$;
        else if $(p \in alive) \wedge (p \in suspected)$ then
            $suspected := suspected \setminus \{p\}$;
            trigger $\langle \Diamond\mathcal{P}, Restore \mid p \rangle$;
        trigger $\langle pl, Send \mid p, [\text{HEARTBEATREQUEST}] \rangle$;
    $alive := \emptyset$;
    $starttimer(delay)$;

upon event $\langle pl, Deliver \mid q, [\text{HEARTBEATREQUEST}] \rangle$ do
    trigger $\langle pl, Send \mid q, [\text{HEARTBEATREPLY}] \rangle$;

upon event $\langle pl, Deliver \mid p, [\text{HEARTBEATREPLY}] \rangle$ do
    $alive := alive \cup \{p\}$;
```

Figure 12: Eventually Perfect Failure Detection pseudocode