5-2019

Consensus in Distributed Systems: RAFT vs CRDTs

Oluwadamilola Okusanya
oookusanya@stcloudstate.edu

Consensus in Distributed Systems: RAFT vs CRDTs

by

Okusanya Oluwadamilola Oluwaseun

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Computer Science

May, 2019

Abstract

Distributed systems normally come with a set of challenges: consistency of data, coordination/synchronization for tasks, failures due to network partitions and so on. Consensus algorithms are a solution to some of these problems especially for coordination and synchronization challenges. These algorithms are normally tied to a consistency model which can fall between two extremes: very strong guarantees on consistency and very weak guarantees on consistency. Two consensus algorithms, RAFT and CRDT are compared using a message queueing system that can form the backbone of a distributed application.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Overview of the Current Research

For a software system to be used by a large cohort of users, in other words *scalable*, it has to be composed of different parts. This extends the life of the software system as the different parts can be changed without devastating side effects to adapt to the varying load the software system would be under. These different parts must work in the system in such a manner that it appears to any user that they are still one monolithic application. These different parts must work in a manner such that the user perceives the software system to be relatively close to himself/herself. In other words, it means that the system must appear to be *location transparent*.

For the software system to achieve goals of scalability and location transparency, it often means that its different parts coordinate themselves in a relatively fast manner in such a way that can be perceived from the user as near instantaneous. This coordination normally happens over some network by the use of messages as the different parts normally reside on different systems. The network used or parts of it might fail causing some of the different parts of the software system to be unable to communicate with each other. This is called *a network partition* and is highly undesirable but commonplace in distributed systems.

A theorem was proposed that encapsulates this problem, the CAP theorem [1]. The theorem can be summarized as follows: a distributed software system cannot be consistent(C), available (A) and have partition tolerance (P). Since most distributed systems strive to be partition tolerant, we are left with two permutations, CP (Consistent + Partition Tolerance) and AP (Availability + Partition Tolerance). CP systems prioritize consistency while AP systems prioritize availability. There are protocols that enable such systems. One of them is Raft [2] which enables a CP system while another is CRDTs [3] [4] which enables an AP system.

1.2 Objective

The goal of this work is to compare a protocol used in a CP system, RAFT, against a protocol used in AP systems, CRDT, in the context of a distributed messaging system. The comparisons will be in the terms of latency and throughput as those are the characteristics on which a distributed messaging system is typically judged [5]. These metrics would be initially judged in the context of a single machine, and if there is time, comparisons using multiple machines would be made.

Chapter 2: RAFT

2.1 Overview of RAFT

Being a protocol used by CP systems, RAFT concerns itself with consistency. It models this problem as a problem of consensus. It tries to present itself as an understandable consensus algorithm, that is, a consensus algorithm that can be easy to implement for people developing highly distributed software systems. This is due to the fact that, according to [2], they struggled to understand the gold standard for consensus algorithms, Paxos [6] [7]. The remark [8] below encapsulates this point

"*The dirty little secret of the NSDI*[1] *community is that at most five people really, truly understand every part of Paxos ;-).*"

RAFT is a leader-based consensus algorithm. It makes use of a single leader for all its decisions. The leader can change depending on circumstances: network partitions, and so on. It models consensus in terms of the replicated state machine architecture [9]. For a network of n nodes to properly handle failures, there must be at least $2f + 1$ servers where f = number of failures. On each of the server nodes in the network, there is a state machine, a log and a consensus module. On each of the client nodes of this network, there is a consensus module. The state machine can be seen as a composition of state-encoding variables and commands that change those variables, thereby changing its state. The logs on the server nodes store the commands from the client nodes in this network, in a particular order which the state machine faithfully follows. It is the responsibility of the consensus module to keep the logs consistent. In order for the system to appear to be highly reliable, these logs are replicas of the master log on the leader, so the state machines at each server are processing the logs in the exact same way. The algorithm used by the consensus module is divided into leader election, log replication and the safety component. In this paper, node and server are used interchangeably and unless otherwise specified, stand for the same thing.

In a cluster of servers that use RAFT, each server at any point in time is in either of these states: *follower, candidate* or *leader*. The modes change based on the RPCs sent between these nodes. A follower is a state where a node is passive, that is, the node cannot issue requests but acts as a recipient for them. A candidate is a transition state where a node in a follower state enters into, on its way to the leader state or a where, a node in a leader state enters into, on its way to the follower state. A follower becomes a candidate by issuing a RequestVoteRPC on its way to being elected a leader. A candidate becomes a follower if it failed to become a leader. A leader is in charge of the cluster and is the only interface to the outside world for the cluster. All communication in the cluster is only between the leader and the followers and is initiated only by the leader.

RAFT does not make any assumption of the state of global synchronization. To this end, the log is partially ordered by a value known as the *term* [2] which increases monotonically. The term can also be infinitely large. This provides some form of synchronization as each server node stores a version of the term on the node itself. Messages in the RPCs carry the source's term. A receiving node processes a received term in two ways:

(a) If the term received is larger than or equal to the receiving node's own local copy, the receiving node overwrites its copy with the received term, and responds with a positive acknowledgement.

(b) If the received term is smaller than the receiving node's own local copy, the receiving node responds with a negative acknowledgement.

2.2 Leader Election

RAFT has a requirement that there should be at most one leader per term. A fixed interval of time is chosen. This interval is used to generate arbitrary values of time. The values are also known as *timeout values* and are used in satisfying the requirement mentioned earlier. These values are randomly allotted to each node except the leader. The generation of *timeout values* and their subsequent allotment are implementation-dependent. After the expiry of a nodes' timeout value, a new timeout value is allotted to it. The process is summarized below in Figure 2.1:

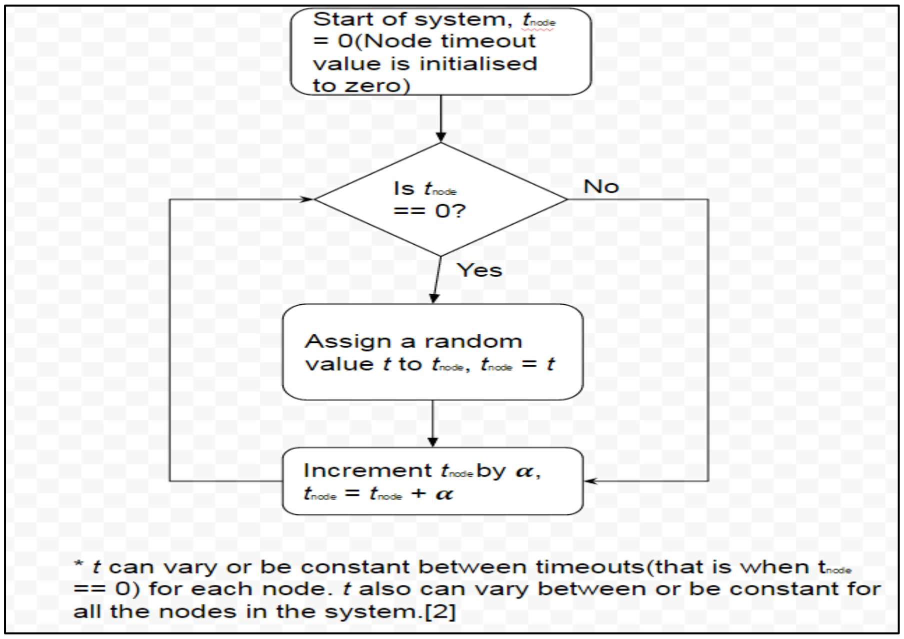

Start of system, t_{node} = 0(Node timeout value is initialised to zero)
Is t_{node} == 0?
No
Yes
Assign a random value t to t_{node}, $t_{node} = t$
Increment t_{node} by α, $t_{node} = t_{node} + \alpha$
* t can vary or be constant between timeouts(that is when t_{node} == 0) for each node. t also can vary between or be constant for all the nodes in the system.[2]

Figure 2.1. Node timeout process (adapted from [2])

α is implementation-dependent and can be *negative* if the cluster has no leader, as a way to elect a leader by determining which node is the fastest to get to 0. α can be *positive* if the cluster already has a leader as a way to prevent the node from reaching 0. The nodes in the cluster get periodic heartbeats from the leader. If the leader is down for a period of time known as the *election timeout*, an election for a new leader is started. The leader election algorithm, based on the assumption that there is no current leader for the cluster, follows these steps [2]:

(a) If the system is starting up for the first time, all the nodes starts out as followers. If any part of this system fails, that part also starts up as followers upon recovery.

(b) If a node times out before the rest of the nodes in the system, it moves from the follower state into the candidate state. This node also increments its term. This node sends out RequestVoteRPCs with its term to every node in the cluster including itself. Figure 2.2 illustrates this process. Here each node's reply (vote) is positive.

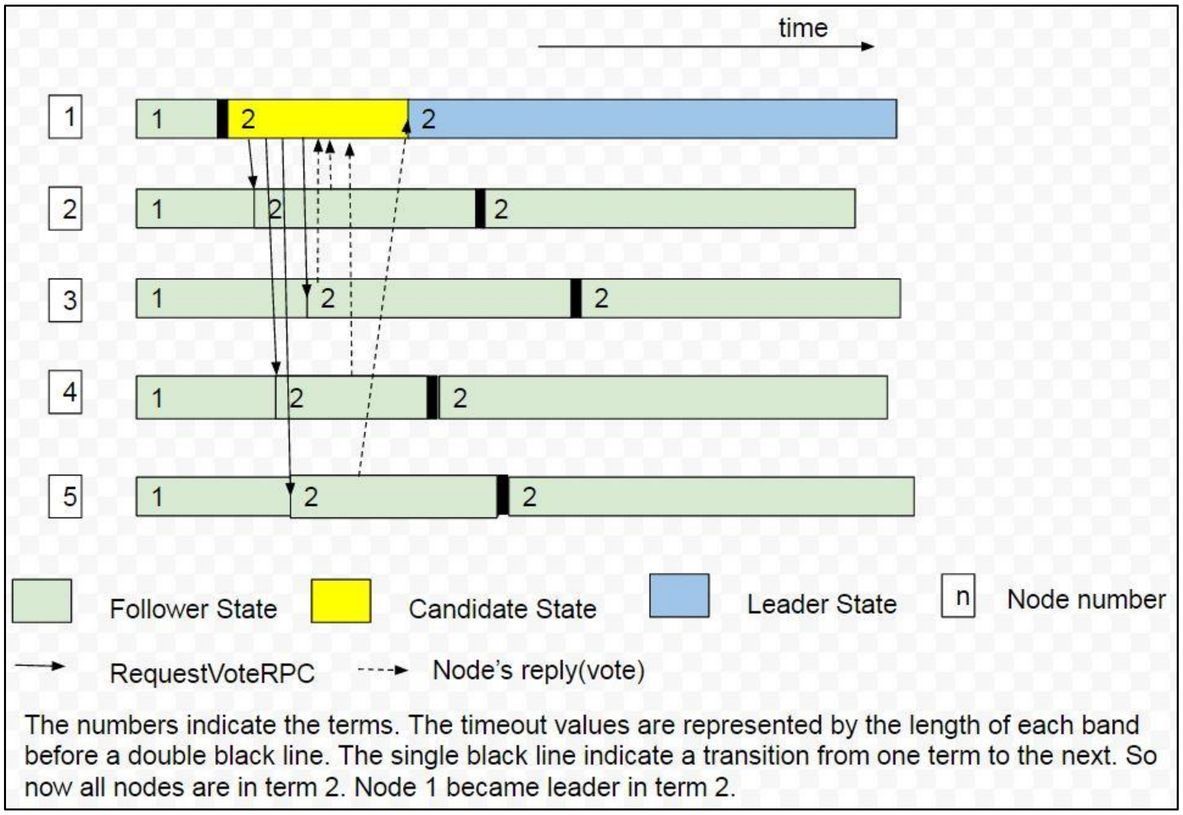

time
1
2
3
4
5
1
2
2
1
2
2
Follower State
Candidate State
Leader State
n
Node number
RequestVoteRPC
Node's reply(vote)
The numbers indicate the terms. The timeout values are represented by the length of each band before a double black line. The single black line indicate a transition from one term to the next. So now all nodes are in term 2. Node 1 became leader in term 2.

Figure 2.2. Successful election (Normal mode) (adapted from [2])

There are times where more than one node timeout at exactly the same time. In this state, the system cannot make progress. This state is called a split move. To resolve this state [2]:

(I) another set of timeout values are immediately calculated and allotted to the affected nodes. This increases the possibility that timeouts would not occur at the same time in the near future.

(II) The timeout values for the other nodes in the system are still decreasing even as the system is in an impasse. At some point, one of the nodes in the system would timeout first and jump to a candidate state or if two or more expire at the same time again, the steps to resolve this state is repeated, that is (I)-(II).

Figure 2.3 shows this process. Here each node's reply (vote) is positive.

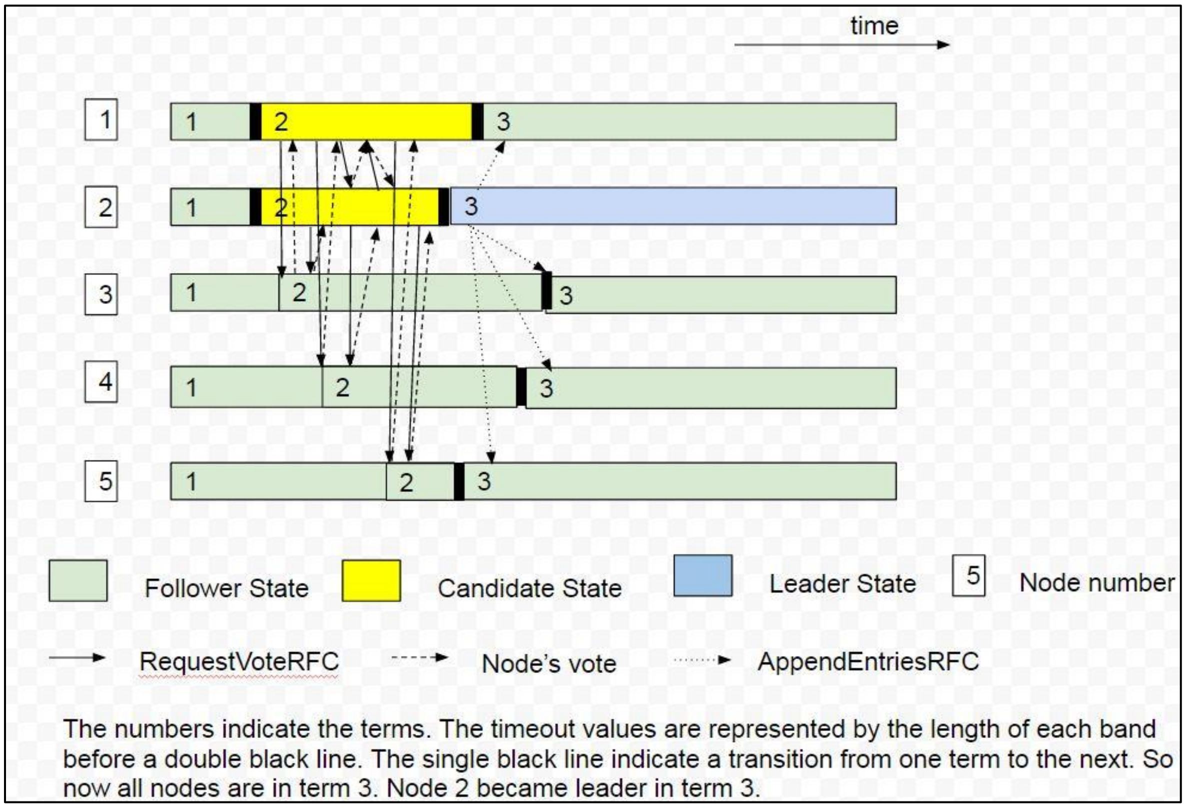

time
1
2
3
4
5
Follower State
Candidate State
Leader State
Node number
RequestVoteRFC
Node's vote
AppendEntriesRFC
The numbers indicate the terms. The timeout values are represented by the length of each band before a double black line. The single black line indicate a transition from one term to the next. So now all nodes are in term 3. Node 2 became leader in term 3.

Figure 2.3. Successful election (Split Vote mode) (adapted from [2])

(c) A node will vote for at most one node to be the leader per term. The only reason a node is unable to vote for a candidate is if there is a network partition or if the node itself has failed. If the node in the candidate state from (b) gets all votes back from a majority of the other nodes in the cluster, the node moves into the leader state and begins to send heartbeats (AppendEntriesRPCs with no log entries) to the other server nodes in the system. These AppendEntries also discourage any would-be nodes from starting its own election process.

(d) Network partitions can occur, where segments of the network are disconnected from the other segments of the network. Consider a scenario where there is a network partition of a system that is composed of five nodes (Node A- E), which divides the system as such; Node A-C in one partition and Node D-E in the other partition:

(i) Node A and Node D timeout and increment their terms. Node A becomes the leader in its partition after receiving votes from B and C, which is a majority of the votes needed (3 out of 5). Node D cannot become leader because it does not receive a strict majority (2 out of 5).

(ii) The partition is removed. Node D timeouts and increments its term and sends out RequestVoteRPCs to all the nodes in the system. They all send their votes back and Node D is elected leader. Note A reverts back to a follower state.

2.3 Log Replication

The log on any server can be viewed as a contiguous collection of entries. Each entry is a combination of a command from a client and the term number of the current leader. There are indexes associated with a log: a commit index for committed entries and a log index for the number of entries in the log. The leader maintains an extra index for every server in the cluster, a *nextIndex*. The difference between the *nextIndex* and the log index is 1. AppendEntriesRFCs that have log entries with their associated log index, are used by the leader to replicate its log among the nodes in the system. Indexes are always incremented by 1. This replication process described in detail below [2] can only be done with a strict majority of nodes.

(a) The leader adds a new entry in its log and increments its log index.

(b) Two log entries are sent out from the leader's log to the follower nodes (one, the log entry of the current log index, *logCurrent* and the other, the preceding log entry, *logBefore*).

(c) Upon receipt of message, the follower nodes check for the following conditions:

- the current term, which is the highest term of the node's log entries must match the received message's term, which is the highest term from the log entries(*logCurrent* and *logBefore*) in the received message.

- the highest index on the node's log must match the log index of the one of the log entries(*logBefore*) in the received message.

If the conditions are met, *logCurrent* is appended to the entry of the follower node and a positive acknowledgement is returned. The leader executes (d). The only cases that the conditions specified above would not hold is if that particular follower node is recovering from a crash or failure. In such cases, the leader executes (e)-(f).

(d) When the leader receives positive acknowledgements from a particular server, the leader increments the *nextIndex* associated with that server. After receiving positive acknowledgements from a strict majority of nodes, the leader commits the entries after applying the commands to its state machine and increments the commit index for every committed entry. The result of the application of the commands in the log entry is returned back to the client.

(e) The leader decrements the *nextIndex* for that entry on the server, assigns the *nextIndex* as the current log index and step (b) is carried out.

(f) (e) is carried out until there is a successful acknowledgement from that particular server. Then the leader can execute step (d).

Figure 2.4 illustrates the above. In Figure 2.4, the *nextIndex* is the index of the next log entry on the leader to be replicated. Figure 2.4(a), (b), (c) represent a normal sequence of operations, that is, they represent a sequence of steps (a), (b), (c) and (d). Figure 2.4(d) shows a scenario where a node (Node 2) has recovered from failure and is missing some entries. Figure 2.4(e) represents a scenario where steps (e) and (f) have been applied to bring down the *nextIndex* for node 2 to a point where replication can occur. Figure 2.4(f) and (g) represents a sequence of operations that allows node 2 to acquire the missing entries from the leader.

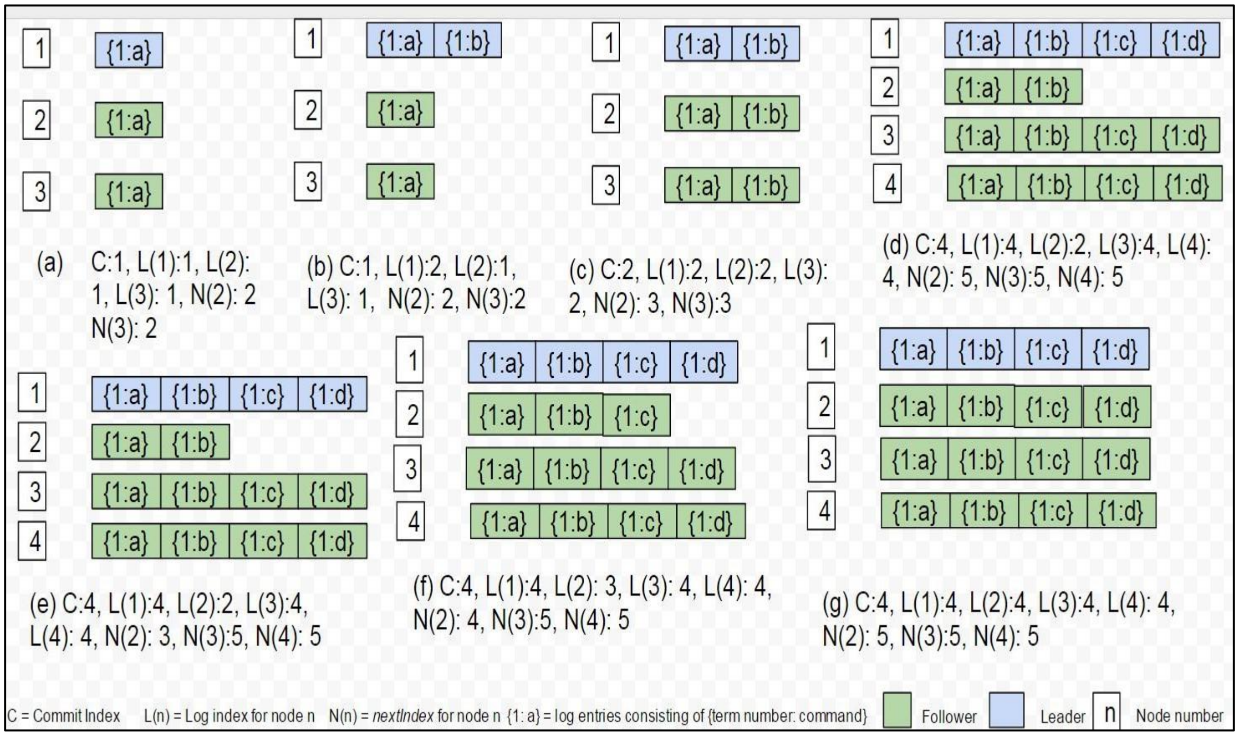

(a) C:1, L(1):1, L(2): 1, L(3): 1, N(2): 2 N(3): 2
(b) C:1, L(1):2, L(2):1, L(3): 1, N(2): 2, N(3):2
(c) C:2, L(1):2, L(2):2, L(3): 2, N(2): 3, N(3):3
(d) C:4, L(1):4, L(2):2, L(3):4, L(4): 4, N(2): 5, N(3):5, N(4): 5
(e) C:4, L(1):4, L(2):2, L(3):4, L(4): 4, N(2): 3, N(3):5, N(4): 5
(f) C:4, L(1):4, L(2): 3, L(3): 4, L(4): 4, N(2): 4, N(3):5, N(4): 5
(g) C:4, L(1):4, L(2):4, L(3):4, L(4): 4, N(2): 5, N(3):5, N(4): 5
{1:a} {1:b} {1:c} {1:d}
C = Commit Index
L(n) = Log index for node n
N(n) = nextIndex for node n
{1: a} = log entries consisting of {term number: command}
Follower
Leader
n Node number

Figure 2.4. Log replication (adapted from [2])

The safety component is used to address certain edge cases in the leader election and the log replication components as shown below [2] by imposing certain restrictions on both the leader election and log replication components of the algorithm.

Table 2.1. Edge cases (adapted from [2])

Edge case	Solution
A node could be elected leader despite not containing all the committed entries.	During the voting phase, a node only returns a positive vote if its last term number <= the candidate's term number and the length of its log <= candidate's length of log.
Future leaders could try to replicate an entry on a formerly crashed leader that was replicated but not committed by the formerly crashed leader. There is a time lag between a replication and a commit. During that time, the leader can crash. So subsequent leaders can try to replicate these entries. This would overwrite the entries stored on the nodes even though those entries have already been stored on a majority of nodes, introducing an unnecessary duplication/redundancy.	Never commit entries from previous terms.

Chapter 3: CRDT

3.1 Overview of the CRDT Protocol

The consistency in the CAP theorem [1] refers to the act of removing accidental non-determinism in computation of values between processes. Consistency tends to be on a sliding scale [10] [11] from strong consistency on one end to weak/eventual consistency on the other end. Strong consistency is normally enforced through synchronization, effectively turning the system into a sequential system.

CRDT is an acronym for Conflict-Free Replicated Data Types. They obey a consistency model called Strong Eventual Consistency [5] which states that in a system, all replicas have equivalent state if and only if the system is eventually consistent and updates have been delivered by those replicas in any order at any node. This means CRDTs are systems that try to achieve consistency without explicit synchronization (synchronization when needed). CRDTs also follow the CALM (Consistency as a Logical Monotonicity) principle [12] which is summarized as a system state should only monotonically increase over time. They also have a property that every operation on them should be commutative, associative and idempotent.

There are two known classes [3] [4]:

(a) State-based CRDTs: They are also called Convergence-Based CRDTs or CvRDTs. States of replicas are exchanged between each other using gossip-protocols. The broadcast operations performed by these protocols to merge states at replicas must also be commutative, associative and idempotent. In other words, they enforce convergence. Convergence here can be thought to have two aspects:

- *Liveness*: a delivery of an updated state to all (interested) nodes in the system happens after a defined number of message exchanges. This happens after a delivery of that same update to a node in the system.

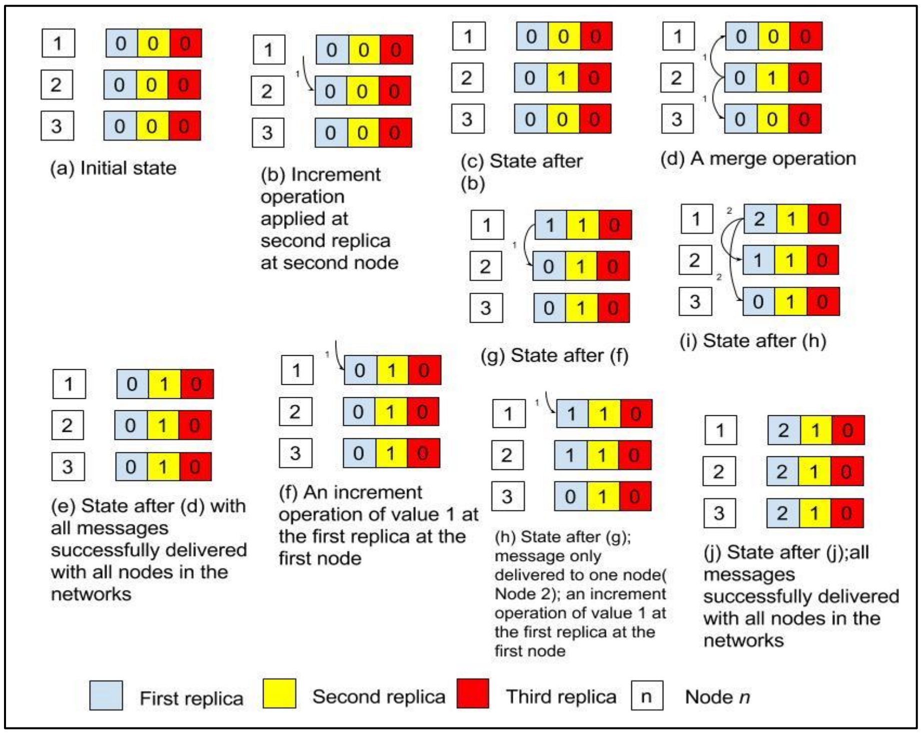
- *Safety*: The same value can be queried from two distinct node that have been delivered the same set of updated states.

(b) Operation-based CRDTs (CmRDTs): They are also called Commutative-Based CRDTs or CmRDTs. In contrast to State-based CRDTs, these only exchange changes in states. They broadcast operations, namely the update performed on them to change their states from one version to another. Sending only these operations reduces the amount of message overhead relative to amount sent by the State-Based CRDTs. They bear remarkable similarities to state machine replication [9] as shown in the following example. Imagine a replica r_1 in a group of replicas which has had update operations u_1 and u_2 applied to it. When it broadcast these operations u_1 and u_2 to the rest of replicas in the group of replicas, these operations u_1 and u_2 can arrive at different times at each replica. To enforce convergence at all replicas in this replica set, these operations u_1 and u_2 must be associative, commutative but not necessarily idempotent. Their idempotency guarantees are based on whatever network protocol is used by the developer to transmit the messages, that is, the ability of CmRDTs to be idempotent is heavily reliant on the reliability of the network protocol used to transmit operations. In the following sections, the primary data structures that are useful for demonstrating and building CRDTs are discussed.

3.2 Counters

A counter [3] [4] is integer-based as it supports only increment and decrement operations. The number of operations can be derived from this integer. A naive CmRDTs version is intuitive, as long as the property that all operations are delivered and applied only once, is rigidly enforced. A naive CvRDT version is not possible as it violates the principle underlying a CvRDT. To illustrate this point, assume that in a set of replicas, the merge operation calculates the max of the values in a subset of replicas. Consider two replicas with an initial state of 0. At each replica, a client executes an increment. Upon a merge of these two replicas, the value would be 1 instead of 2 as there were two independent increment operations, one for each of the replicas. Based on this observation, we must modify the naive CvRDT counter.

Assume we use an integer vector, with the size of the vector equal to the number of replicas in the system. Each index in this vector corresponds to a particular replica. The merge operation can compute the max by getting the maximum value of each position in the vector. This whole procedure is explained in Figure 3.1:



(a) Initial state
(b) Increment operation applied at second replica at second node
(c) State after (b)
(d) A merge operation
(e) State after (d) with all messages successfully delivered with all nodes in the networks
(f) An increment operation of value 1 at the first replica at the first node
(g) State after (f)
(h) State after (g); message only delivered to one node(Node 2); an increment operation of value 1 at the first replica at the first node
(i) State after (h)
(j) State after (j);all messages successfully delivered with all nodes in the networks
First replica
Second replica
Third replica
Node n

Figure 3.1. An increment-only CvRDT counter (adapted from [3])

To support decrements, another type of counter, a PN Counter [3] [4] is used. In a PN Counter, two counters are associated with each element and it tracks the element modification: the deletion and addition of that element the counter is associated with. A decrement is an increment on the counter that is associated with deletion. The same goes for the increment operation. A query is the value of difference of the two counters. An example of a PN Counter is shown below:

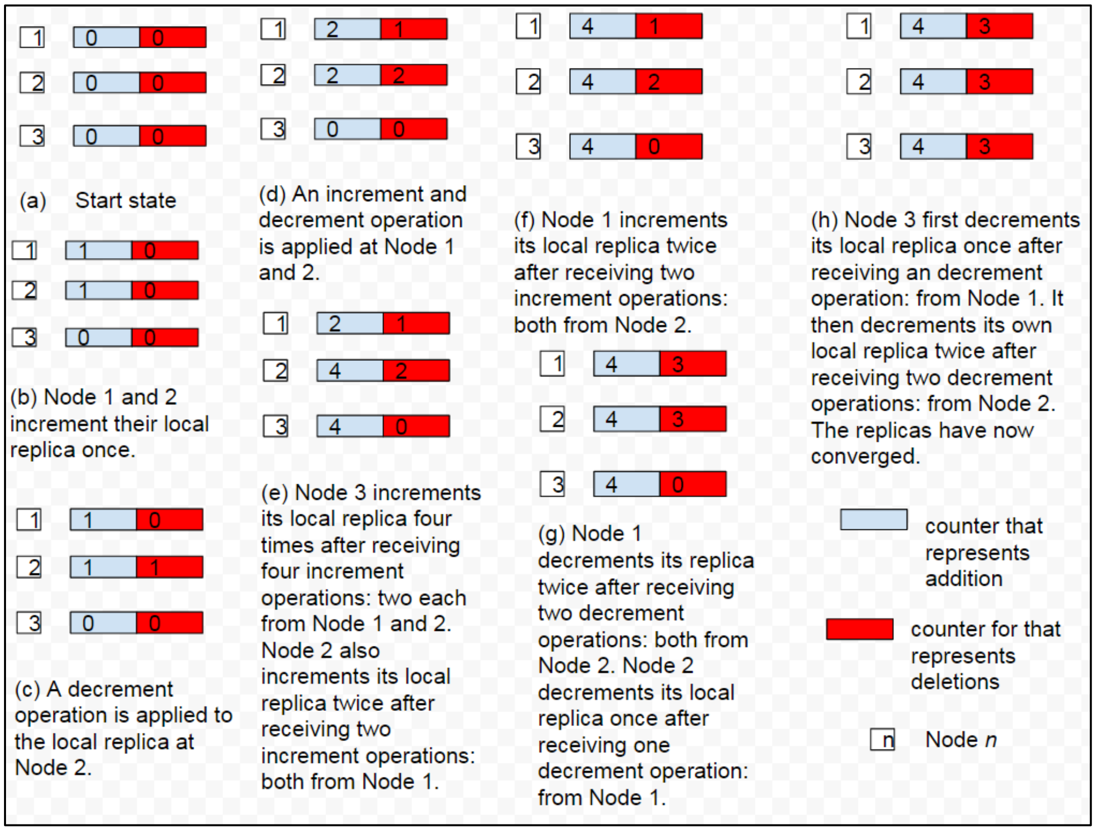

(a) Start state
(b) Node 1 and 2 increment their local replica once.
(c) A decrement operation is applied to the local replica at Node 2.
(d) An increment and decrement operation is applied at Node 1 and 2.
(e) Node 3 increments its local replica four times after receiving four increment operations: two each from Node 1 and 2. Node 2 also increments its local replica twice after receiving two increment operations: both from Node 1.
(f) Node 1 increments its local replica twice after receiving two increment operations: both from Node 2.
(g) Node 1 decrements its replica twice after receiving two decrement operations: both from Node 2. Node 2 decrements its local replica once after receiving one decrement operation: from Node 1.
(h) Node 3 first decrements its local replica once after receiving an decrement operation: from Node 1. It then decrements its own local replica twice after receiving two decrement operations: from Node 2. The replicas have now converged.
counter that represents addition
counter for that represents deletions
n Node *n*

Figure 3.2. A PN counter (adapted from [3])

Figure 3.2 illustrates a CmRDT-based PN counter. Here a value has three replicas stored at three nodes, one replica for each node. Additions and deletions to this element are modeled with two counters, one for additions, and the other for deletions. Additions and deletions add 1 or remove 1 from the value respectively, but are both represented as increments to the counters representing them. Figure 3.2(a)–(h) show the states at each node after a particular operation or sequence of operations have been applied. In Figure 3.2(d), the operations are applied as such, increment first and then decrement.

3.3 Registers

A register [3] [4] acts as a placeholder for objects of a particular type. The two operations on this data structure are *assign* and *value*. Implementing CRDTs semantics using this data structure is tricky but there are two known implementations [3] [4]:

(a) Last Write Wins Register (LWW): The values are ordered by an increasing value, usually a timestamp. These timestamps are assumed to possess properties of uniqueness, total ordering and casual ordering. The current value is retrieved using the *value* operation. For CvRDTs of this type of register, the *assign* operation updates the values contained in the register and generates a new timestamp to be associated with the operation. The *merge* operation returns the value that has the maximum timestamp associated with it. For CmRDTs of this type of register, the *assign* operation generates a new timestamp at the sender. The receiver only updates the value if the new timestamp is greater than its own timestamp for that particular value.

(b) Multi Value Register (MV): Here values are stored as set of pairs (value, version vector [13]) for example an object in this system is stored in this way:{a_1: unique identifier for replica a_1, a_2: unique identifier for replica $a_2, \dots ,a_n$: unique identifier for replica a_n} where n is the number of replicas for that particular object. These unique identifiers are usually composed of a timestamp and a monotonically increasing value to implement some sort of ordering. Version vectors summarize the history of updates to a particular replica. Version vectors have been shown to track causality [13] in distributed systems. An example of a version vector is shown below. For version vector *A* and *B* for an object *F*, we can say that

(I) *A* dominates *B* if *A* > *B*, that is every element of *B* is less than or equal to the corresponding element of *A* and at least of one of the elements of *B* is strictly less than corresponding element of *B*.

A = [{a, 1}, {b, 2}, {c, 3}]

B = [{a, 4}, {b, 2}, {c, 3}]

B dominates *A*.

(II) *A* descends *B* if *A* >= *B*. A summary of the occurrence of the events in the system is achieved with this method. For example:

A = []

B = [{a, 1}]

B descends *A* as all version vectors descends the empty vector.

(III) *A* is concurrent or in conflict with *B*, that is, some elements of *A* are not in *B* and some elements in *B* are not in *A*. Also *A* conflicts with *B* if two or more elements do not represent a monotonically increasing order. For example:

A = [{a, 1}]

B = [{b, 1}]

A is concurrent with *B*.

A = [{a, 1}, {b, 2}, {c, 4}, {d, 3}]

B = [{a, 1}, {b, 2}, {c, 3}, {d, 4}]

A conflicts with *B*.

Everything we have discussed so far concerning Multivalue registers have been for CvRDTs. Operations on this CRDT are

(a) *value*: read a local copy of vector

(b) *assign*: This is equivalent of a write. It generates a vector that dominates the previous ones.

(c) *merge*: This takes the pairwise maximum of every element of the vectors being compared.

An example is shown in Figure 3.3:

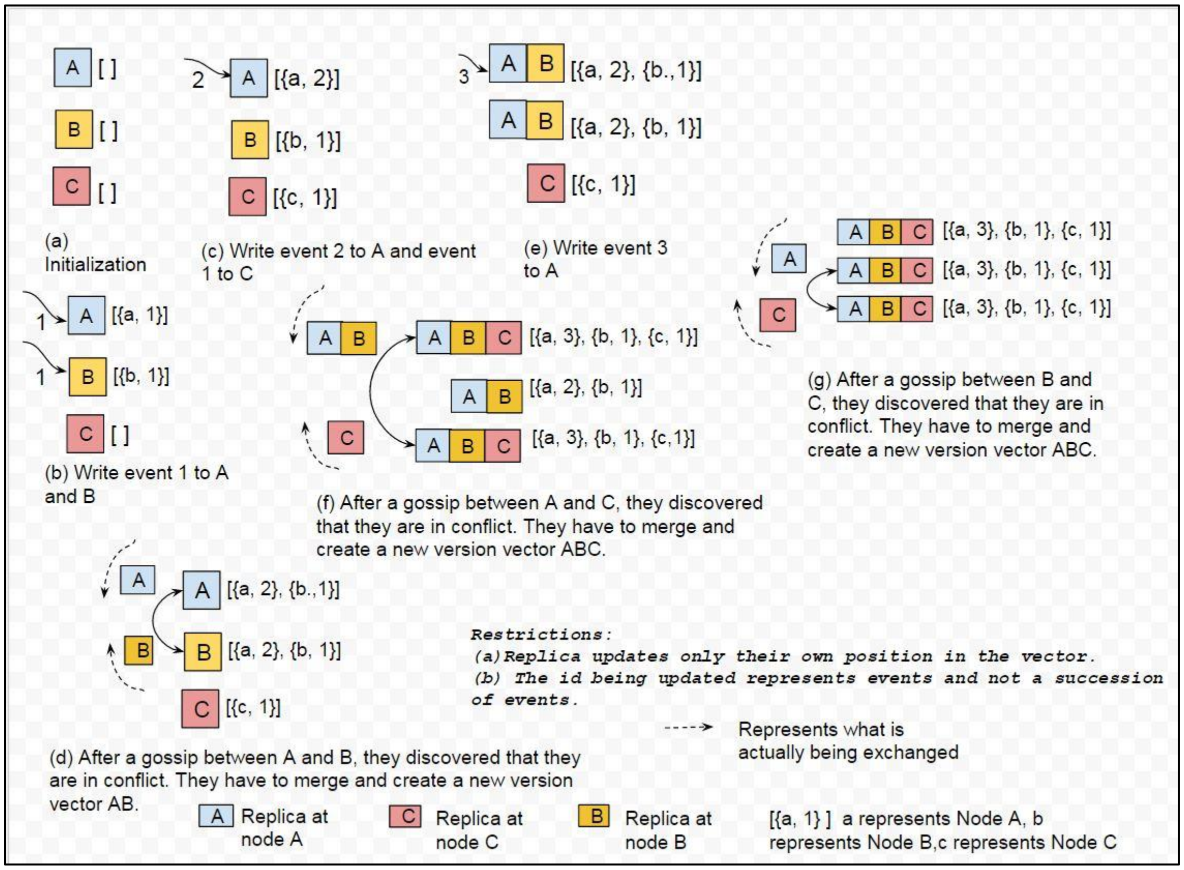

A [ ]
B [ ]
C [ ]
(a) Initialization
2 A [{a, 2}]
B [{b, 1}]
C [{c, 1}]
(c) Write event 2 to A and event 1 to C
3 A B [{a, 2}, {b.,1}]
A B [{a, 2}, {b, 1}]
C [{c, 1}]
(e) Write event 3 to A
1 A [{a, 1}]
1 B [{b, 1}]
C [ ]
(b) Write event 1 to A and B
A B
A B C [{a, 3}, {b, 1}, {c, 1}]
A B [{a, 2}, {b, 1}]
C
A B C [{a, 3}, {b, 1}, {c,1}]
(f) After a gossip between A and C, they discovered that they are in conflict. They have to merge and create a new version vector ABC.
A
C
A B C [{a, 3}, {b, 1}, {c, 1}]
A B C [{a, 3}, {b, 1}, {c, 1}]
A B C [{a, 3}, {b, 1}, {c, 1}]
(g) After a gossip between B and C, they discovered that they are in conflict. They have to merge and create a new version vector ABC.
A
A [{a, 2}, {b.,1}]
B
B [{a, 2}, {b, 1}]
C [{c, 1}]
(d) After a gossip between A and B, they discovered that they are in conflict. They have to merge and create a new version vector AB.
Restrictions:
(a)Replica updates only their own position in the vector.
(b) The id being updated represents events and not a succession of events.
Represents what is actually being exchanged
A Replica at node A
C Replica at node C
B Replica at node B
[{a, 1}] a represents Node A, b represents Node B,c represents Node C

Figure 3.3. A multi value register (adapted from [3])

The use of version vectors comes with a caveat: they have unbounded growth. In this area, various improvements [14] have been proposed.

3.4 Sets

Implementing a set [3] [4] naively for CRDTs conflicts with the definition of a CRDT. Consider two operations; *insert* and *delete.* A union is the result of an insert and a minus is the result of a delete. Assume that we are dealing with a CmRDT based set. Assume we have a system composed of three replicas: *a, b, c*. We apply operations as they come to each replica in causal order. A hypothetical sequence of operations could be as follows:

Replica *a* insert a value 2 into its set. State of its set = {2}
Replica *b* insert the same value 2 into its set. State of its set = {2}
Replica *a* receives the insert operation from Replica *b* and inserts the same value 2 into its set. But since it is a set, the operation has no effect.
Replica *a* removes the value 2 from its set. State of its set = {}
Replica *b* receives the insert operation from Replica *a* and inserts the same value 2 into its set. But since it is a set, the operation has no effect.
Replica *b* receives the remove operation from Replica *a* and removes the value 2 from its set. State of its set = {}
Replica *c* receives the insert operation from Replica *a* and insert the same value 2 into its set. State of its set = {2}
Replica *c* receives the remove operation from Replica *a* and removes the value 2 from its set. State of its set = {}
Replica *c* receives the insert operation from Replica *b* and inserts the same value 2 into its set. State of the set = {2}
The final states of the replicas are Replica *a* = {}, Replica *b* = {}, Replica *c* = {2}.

Replica *c* has not converged, that is, its state is not the same as that of Replica *a* and *b*. This shows that using sets as we know them from mathematics, without some modifications or conditions attached, would invalidate the preconditions of a CRDT: *associativity*, *commutativity*. One modification to allow the use of sets as CRDTs has been based on whether or not they support the remove (minus) operation. The set-based CRDTs also differ on how the remove operation is implemented if the remove operation is supported.

(a) G-set [3] [4]: This set-based CRDT does not support the remove function. The operations available are *add* and *value*. The *add* operation for CvRDTs of this type are state modifiers if there is some causal order on the states. The *add* operation is commutative as it is implemented as a union for CmRDTs. For example {1} U {2, 3} = {2, 3} U {1}. This type is not very useful on its own. It has a similar operation to the increment only counter discussed earlier.

(b) 2P set [3] [4]: This supports the remove operation in the form of *tombstones*. Basically, the 2P set has two G-sets, one for the insertion G_i and the other for deletion G_d. G_d is a tombstone set. An element x can be removed from G_i *if and only if* $x \in G_i$. To remove an element from the set is to place it in G_d. Once placed in G_d, it cannot be added back again to G_i. For 2P CvRDTs, the *value* operation returns elements that have not been removed. The *merge* operation returns the union of the two sets, G_i and G_d. For 2P CmRDTs, the *remove* and *add* operations commute, that they are inherently commutative with a slight modification: a *remove* operation only occurs after an *add* [3] [4].

Chapter 4: Methodology

These protocols are used in the building of distributed systems. An example of a distributed system in the terms of this paper is a messaging system.\

Current large enterprise software in the cloud are becoming *microservices*-based [15]. A *microservice* [15] is a modular approach to building software where different parts of the application are split on business function(*services*) and communicate over HTTP or RPC. The connections between these *services* can be modeled as a producer which generates data and a consumer that processes such data to give results and a pipeline between them. This producer and consumer might be in the same enterprise software or they might be in different enterprise software. The pipeline brings the producer and consumer together as it transfers the data between the producer and consumer. This pipeline might be asynchronous (a message queue) or synchronous (an RPC protocol).

A message queue is the main part of a messaging system which is normally comprised of the queue, messaging protocol and endpoints which are in this case, a producer and consumer. Depending on the messaging system, most of the queues are set up as a broker which transforms the format of the message sent by the producer into the format accepted by the consumer. Figure 4.1 gives us a brief overview of such the messaging system described above. Most times, brokers do not have transactional support. Transactional support in this case means sending and receiving a message is treated as a transaction. The sender sends a message to the message queue. If the receiver(s) has/have not received this message and acted upon it, then the message is not deleted from the message queue. This work hopes to use the protocols discussed in the earlier sections (RAFT, CRDTs) to enable transactional support in a message queue, through some form of distributed consensus.

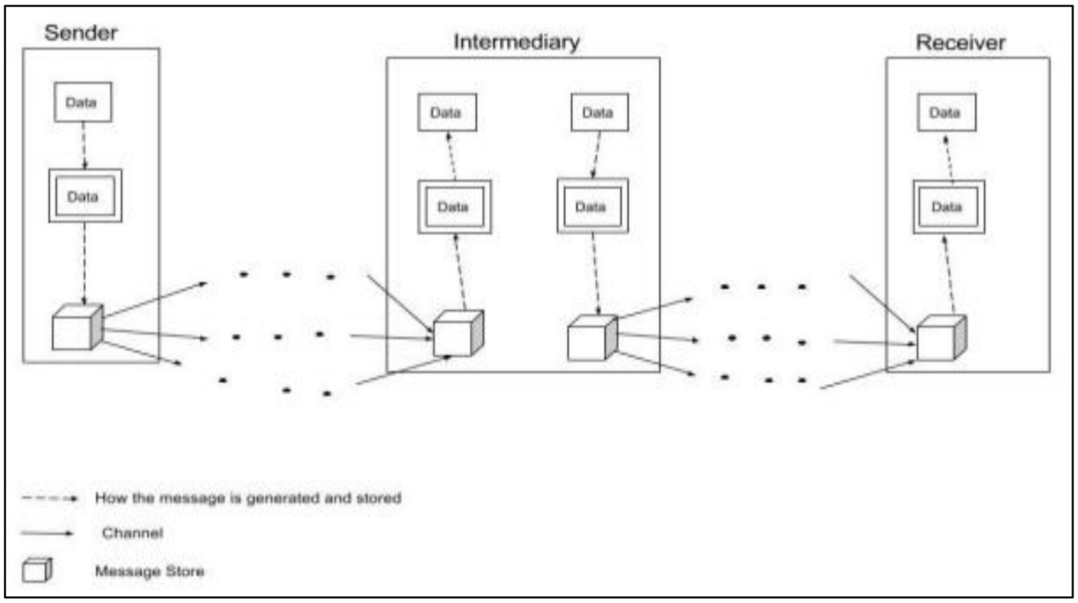


Figure 4.1. A general messaging system

4.1 High Level Overview of the Message Queue

The message queue is modeled as a broker. The broker is responsible for receiving and routing messages from sender to receiver. The broker here is actually a cluster of nodes/servers. Consumers and subscribers are used interchangeably in this section and unless specified, mean the same thing.

4.2 RAFT

Here each node in the cluster has the same configuration as Figure 4.2 (without the Producers and Consumers). For the nodes that are not the leader, the Producers and Consumers section of Figure 4.2 is replaced by the leader. Only the leader in the RAFT model has this cluster manager/sharing layer enabled.

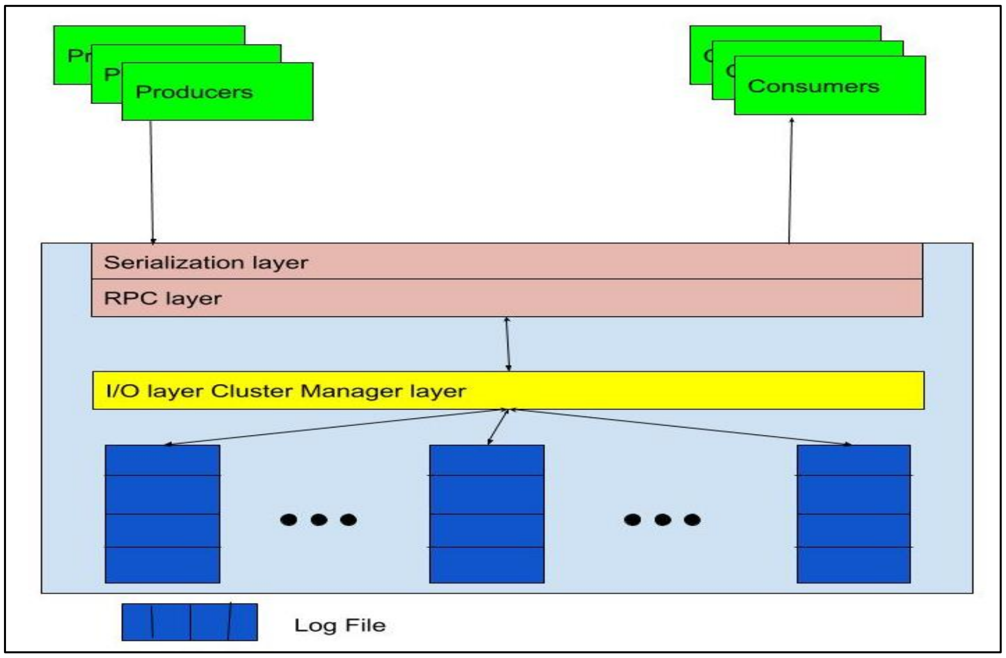


Figure 4.2. Application architecture of the leader in the broker-RAFT

The node is composed of two components:

(a) Middleware component: This is the layer that actually interacts with the servers and clients. It is composed of a:

- *Serialization layer* for serializing the data to a binary format for compact storage.

- *RPC layer*. This layer enables communications between different nodes.

(b) Consensus component: This layer is what implements the consensus protocol. It is divided into two:

- *Log files*: A log file represents a topic. It is a file on the disk that stores the messages received from the producer. Messages are appended at the end of the file. Messages are read by the I/O layer from the beginning of the file to some offset *n* based on some mapping in the I/O layer. Messages are only removed from a log file after all of the available subscribers have received the message and sent back positive acknowledgements.

- *A cluster manager*: The cluster manager is responsible for reducing/increasing the number of nodes that are required based on the following metrics: message volume load from the producers and cpu metrics of the system running the broker.

- *A I/O layer:* The I/O manager is responsible for sending the messages received from the producer based on the topic id into the different log files. The I/O layer is also responsible for deciding the number of messages *n*, to send to subscriber S_i in subscriber group *S*. This value *n* is equal to the *n* described in (a). There can be multiple subscribers for a particular topic. To scale properly on the subscriber side, each subscriber S_i is fitted with a cache that they can poll messages from. The cache eases pressure on the message queue and provides an asynchronous way of handling messaging. Also, the I/O layer maintains a mapping {S_i, *n*} for all the subscribers in *S*. Changes to this mapping are replicated using the RAFT protocol.

A message will be of the form {*topic-id*, *value*}. When the leader gets a message from a producer, it appends the *value* of the message to the relevant log file based on the *topic-id* associated with the log file. The leader then sends the message out to the all the nodes in the cluster. Each node appends the *value* of the message received from the leader to the relevant log file based on the *topic-id* associated with the log file and returns a positive acknowledgement to the leader. To remove a message(s) stored on the leader, the leader sends a remove request {*topic-id*, *n*} to each node in the cluster. Each node physically removes the *n* messages from the relevant log file according to the *topic-id* associated with the log file and returns a positive acknowledgement to the leader. The leader then proceeds to remove *n* messages from the relevant log file according to the *topic-id* associated with the log file.

4.3 CRDT

For the CRDT model, Figure 4.3 is the representation of the broker. The broker here is a cluster of nodes/servers. There are two types of nodes in this broker, a leader node and peer node. There are always two leader nodes in the system. One is the master and the other is the slave. The master is the leader node that handles all communications between the broker and the outside world (producers and consumers) while the slave does not partake in any of the communications in the system. If the master leader node fails, the slave node becomes the new leader node and a copy of the slave node is created and becomes the new slave node. The leader nodes do not store any messages; the peer nodes do. Also, the I/O layer is unique to the leader nodes as is the peer-to-peer layer for the peer nodes.

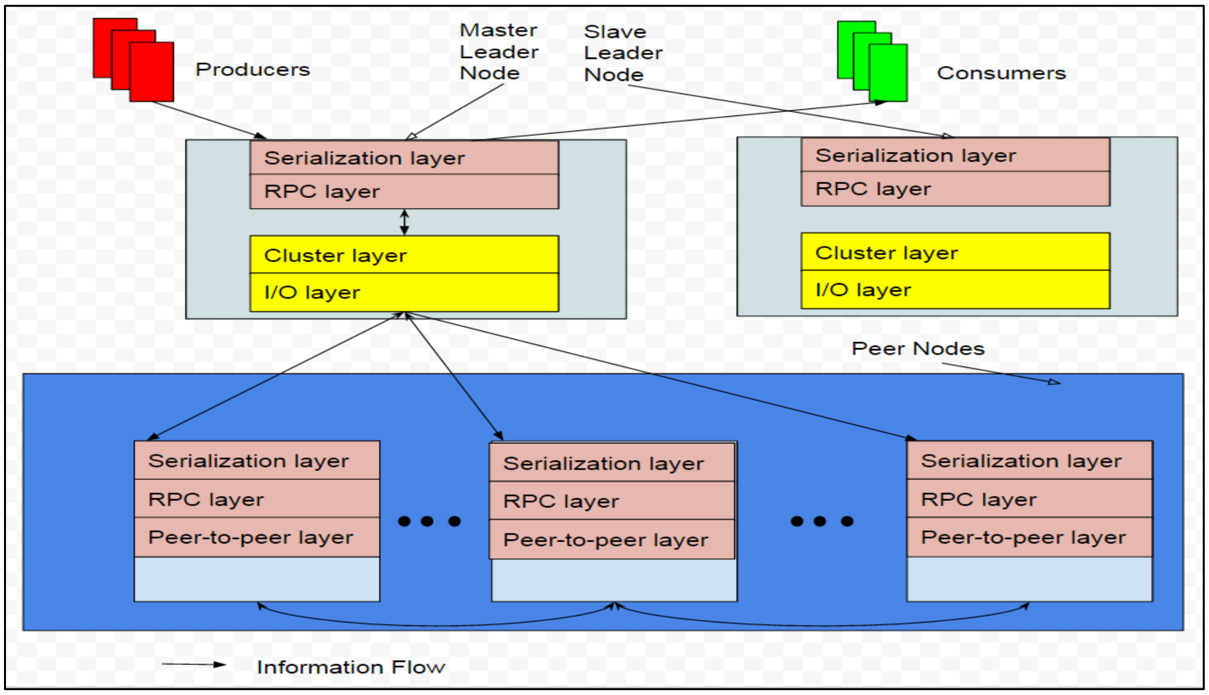


Figure 4.3. The broker application architecture for the CRDT model

The serialization layer, RPC layer are the same as the ones described in the RAFT model. Here the cluster layer is the same as the cluster manager layer in RAFT. The I/O layer here is responsible for inserting and retrieving the messages stored on all the peer nodes. It inserts messages into nodes randomly. The CRDT responsible for holding the message in Figure 4.3 is a map [16]. A map is a structure similar to a regular map data structure in that it exposes similar operations such as get(key) and put (key, value). It is different from regular map operations as the keys and values are CRDT values. The keys in this case are LWW (Last Write Wins) registers. The values are sets. The sets in this case are LWW-sets [3] [4]. The LWW-set is similar in operation to the 2P-set in that it has an add-component (set) and a remove-component(set). In the LWW-set, a pair (timestamp, visibility) is associated with a value. A visibility of true denotes an add while a visibility of false denotes a remove. To remove an element, you add (value, (element timestamp, false)) to the remove set and to add an element, you add (value, (element timestamp, true)) to the add set. A merge operation is done by merging the add and remove sets together. A lookup for element *e* results in true value if and only if the element *e* is in the add set and not in the remove set with a higher timestamp, that is, the element *e* in the add set must have a higher timestamp than a corresponding element *e* in the remove set. This also holds even if the element *e* has duplicates in the add set. In the event of a concurrent add and remove, the removes only occur after the adds. Timestamps increase monotonically.

A description of the maps that are used by the brokers are shown below:

```
map (service or user) {

        register: service-name;

        register: user-name;

        set: elements

}
```

After receiving new messages from the leader, the peer-to-peer layer of a peer node is responsible for sending out updates to the other nodes in the cluster. An update operation is performed at a replica by merging the maps' sets together.

When the consumer issues a read of *n* elements from the broker, that is the consumer issues a *get*(*key(s), n*) command, these steps are followed in sequence:

(a) each replica is contacted by the I/O layer to retrieve n elements from the map whose keys match with the keys(s) in the command.

(b) each replica performs a merge of the two sets (add and remove) of all the identified maps present in the replica.

(c) the peer-to-peer layer of each replica sends out messages containing the merged sets to all peers

(d) a merge operation is then conducted at each replica.

(e) all the replica's sets are then merged to produce a set *A*.

(f) The lookup operation is run on each element in set A and returns either a true or false value. Those elements that return true, are returned from the broker and pulled into the customer cache. Steps (g) –(j) are done after a positive acknowledgment is returned from the customer. Steps (g) – (j) are necessary to keep the data stored at a manageable level.

(g) A remove operation, *remove(key(s), n)* is then issued by the I/O layer to a replica. Assuming that an *add ()* operation has just been applied to the replica, the I/O layer takes note of the timestamp *t* issued by the *add ()* operation and removes n elements associated with key(s) whose timestamp is less than that of *t*. Otherwise, the I/O layer removes *n* elements associated with key(s). In both scenarios, those elements are added into the remove set of that replica.

(h) The peer-to-peer layer then replicates this remove set with all the replicas in the set.

(i) When (h) is complete, a *removeElement()* operation is performed at each replica. This removes each element in the add set that is also in the remove set.

(j) After that, each element in the remove set is purged.

Chapter 5: Implementation

This project uses a few libraries and technologies. These technologies/libraries are as follows:

- Docker
- ZMQ

In the sections below, explanations of how these libraries are used in this project will be given. Also, the RAFT/CRDT configurations using the aforementioned libraries will also be discussed.

5.1 Docker

Docker is a library that provides a lightweight VM. It is a system-level VM meaning that it houses the system-level libraries in order to run the simulation. It does so, by making use of resource isolation (that is address space isolation, network isolation, I/O isolation, memory isolation) found in the Linux kernel mainly *cgroups* and *namespaces* [17]. An example comparison between Docker and the normal type of VM's such as VirtualBox is shown in Figure 5.1:

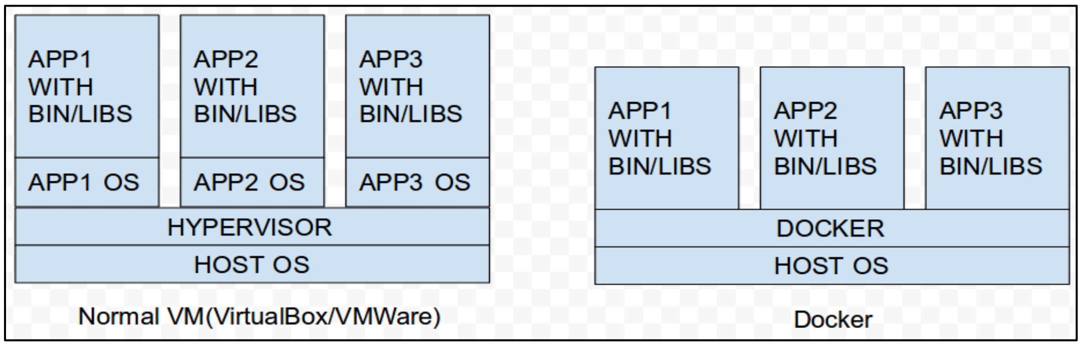


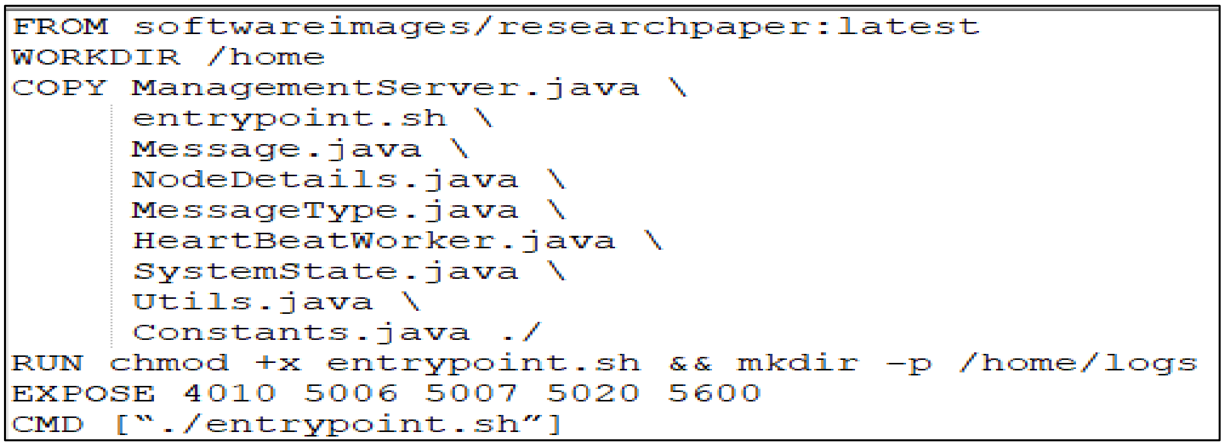
Figure 5.1. Comparison of Docker vs VirtualBox [18]

The main difference between Docker and a normal VM (VirtualBox) are that a normal VM provides a system-level, system-type resource isolation meaning that they provide total isolation between two systems (that is, two host systems running on the same machine) and Docker provides a system-level, process-type/application-type resource isolation, meaning that only the applications/process are separated but share the same underlying system. This makes Docker useful for this project as:

(a) Each process can be packaged into a *container*, instances of an *image* and with the system-level support can be regarded as a machine. When a set of these machines are put on a network, they become a *distributed system*.

(b) With isolation you can transplant a particular bunch of containers onto another host system and it will still produce the same output, providing *repeatability*.

An *image* is a snapshot of a system with system-level libraries that provide the perfect environment to run a process. This *image* is immutable and it has a *layered* filesystem [17] [19]. A *container* is a running/stopped instance of a image that is *namespaced* [17] [19], that is it can only use kernel resources (network, file, user and so on) associated with that namespace. To create an image, a Dockerfile is needed. A sample Dockerfile for this project is shown in Figure 5.2.

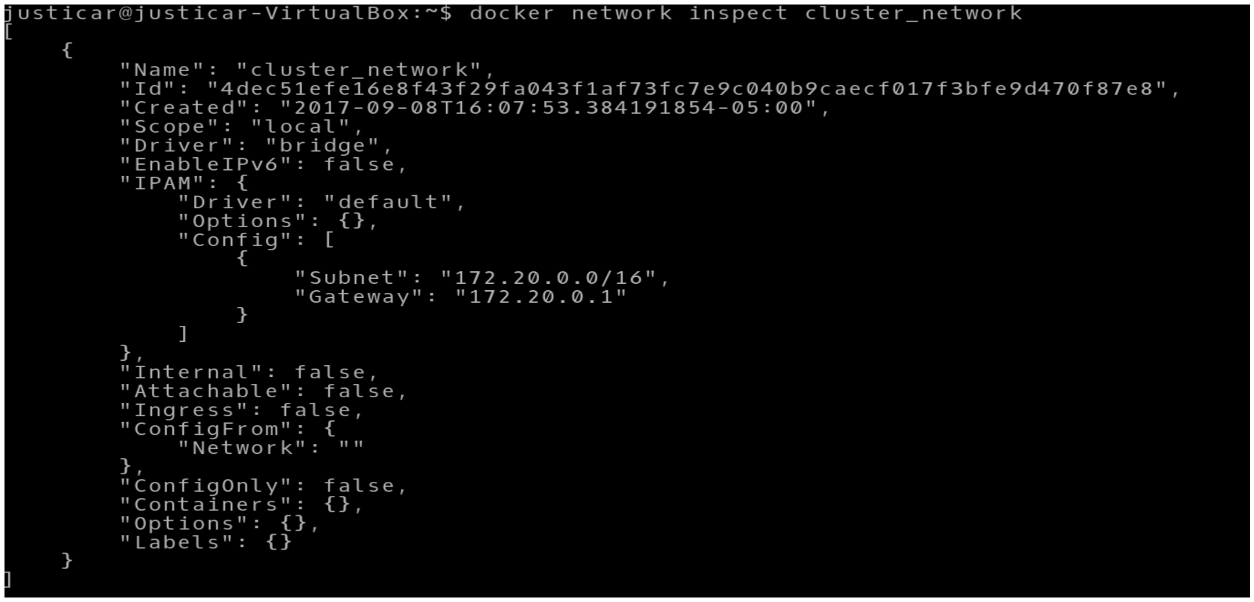


```
FROM softwareimages/researchpaper:latest
WORKDIR /home
COPY ManagementServer.java \
     entrypoint.sh \
     Message.java \
     NodeDetails.java \
     MessageType.java \
     HeartBeatWorker.java \
     SystemState.java \
     Utils.java \
     Constants.java ./
RUN chmod +x entrypoint.sh && mkdir -p /home/logs
EXPOSE 4010 5006 5007 5020 5600
CMD ["./entrypoint.sh"]
```

Figure 5.2. A Dockerfile for the management server

Figure 5.2 contains instructions to generate the image that would contain all the files necessary to run a process and it is built from another image: *softwareimages/researchpaper: latest*. The dockerfile for that image is found in Appendix A. A complete meaning of the commands used is also explained in Appendix A. To generate the Docker containers from the images, a native command is used: *docker run.* To automate generation of Docker containers from images, another native command *docker compose* is used for that purpose.

In order for containers to talk to each other, Docker provides the capability of creating a virtual network. A virtual Ethernet *bridge* is created for the user when the user installs Docker. This *bridge* allows for the communication between containers without the use of port forwarding, as this *bridge* is a self-contained IP subnet and gateway. This *bridge* also supports automatic DNS name resolution, resolving the container names to the IP addresses assigned by the *bridge*. This bridge is local. Figure 5.3 shows the configuration of the network used for the simulation.



```
justicar@justicar-VirtualBox:~$ docker network inspect cluster_network
[
    {
        "Name": "cluster_network",
        "Id": "4dec51efe16e8f43f29fa043f1af73fc7e9c040b9caecf017f3bfe9d470f87e8",
        "Created": "2017-09-08T16:07:53.384191854-05:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.20.0.0/16",
                    "Gateway": "172.20.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Figure 5.3. The configuration of the network, *cluster_network*

5.2 ZMQ

ZMQ/ OMQ (pronounced zero-m-q) [20] is a networking library. It is a wrapper around the C socket library and provides easier ways of creating and utilizing network sockets. The library has several language bindings including a Java binding named *jzmq*. The library is based around *ZMQ* sockets which at their core are composed of two things: a C socket and a queue. The queue allows for the library to send message when it wants to and also allows for the retaining of messages that were not delivered to their destination. ZMQ is also built on the idea that normal lock-based synchronization primitives are too error-prone and brittle to be used effectively and proposes a message-only solution for synchronization [20].

A major benefit of *ZMQ/ OMQ* is that it allows for specific patterns to be used very easily which forms the different types of sockets to be found in this project. These types of sockets are [20]:

- *REQ* socket (a receive-type)
- *REP* socket (a send-type)
- *DEALER* socket (a receive-type),
- *ROUTER* socket (a send-type),
- *PAIR* socket (a receive-type and send-type)
- *PUB* socket (a send-type)
- *SUB* socket (a receive-type)

These sockets are best understood in pairs [20]:

(a) A REQ/REP pair is the basic client/server connection type and is the basis of all the other subsequent types of sockets that are listed above.

(b) A DEALER/ROUTER pair is analogous to the REQ/REP pair. The difference between the REQ/REP pair and the DEALER/ROUTER pair is that the DEALER/ROUTER is primarily used for asynchronous connections while the REQ/REP pair is synchronous.

(c) A PAIR/PAIR pair is primarily used between threads of the same process.

The main differences between them can be summarized in Table 5.1:

Table 5.1. Differences between ZMQ Sockets (adapted from [21])

Socket type	**Send**	**Receive**	**Fixed-order**	**Socket placement**	
				Client-side	Server-side
REQ	Yes	No	Yes. REQ can only receive after it has sent a message.	Yes	No
REP	No	Yes	Yes. REP can only send after it has received a message.	No	Yes
DEALER	Yes	Yes	No. DEALER can send or receive at any time in any order.	Yes	No
ROUTER	Yes	Yes	No. ROUTER can send or receive at any time in any order.	No	Yes
PUB	Yes	No	Yes. PUB can only send messages.	No	Yes
SUB	No	Yes	Yes. SUB can only receive messages	Yes	No
PAIR	Yes	Yes	No. PAIR can send or receive at any time in any order.	Yes	Yes

There are certain caveats which were discovered when using this library in building this project. They are:

- *ZMQ/ OMQ* is an *async* framework, meaning that it expects the user not to depend heavily on timing and order of arrival. This reorients the architecture of the network and makes the user to have to deal with messages that have long since been delivered which are not necessary at a certain stage of the computation, that is, duplicates of an already processed message can arrive at a receiver.

- *ZMQ/ OMQ* does not send instantaneously. A send command hands off the message from the user process to the underlying ZMQ thread/process which sends the message on its own time. It is a fire-forget mechanism meaning that depending on the type of socket, the message will be dropped if the destination is unreachable.

- *ZMQ/ OMQ* sockets are not thread-safe. They are made thread-safe when used with a *ZMQ Context* [21]. This does not allow them to be passed among multiple processes, that is, a *ZMQ* socket exhibits undefined behavior [20]. The process that it is attached to (the owner of the socket) is the only one that can use the socket. This is keeping inline with their message-only philosophy discussed earlier.

(d) Based on the socket semantics as described above, it is important to divide the network into static parts. That is, one would typically have the *REP/ROUTER* sockets bound to the “server” parts of the network and the REQ/DEALER sockets bound to the “client” parts of the network.

All these caveats/properties were considered in the design of the network architecture for both the RAFT and CRDT configurations.

5.3 RAFT

5.3.1 Overall system architecture

The overall socket/docker/system architecture is shown in Figure 5.4. Each box contains a *ZMQ Poller* which uses *select()* and *poll()* internally to deque messages received by the sockets to be made available for use in the program. Every socket in the box is registered to a *ZMQ Poller*. From here on out, every time the word *container* with respect to Figure 5.4 is used, it represents a box in that image.

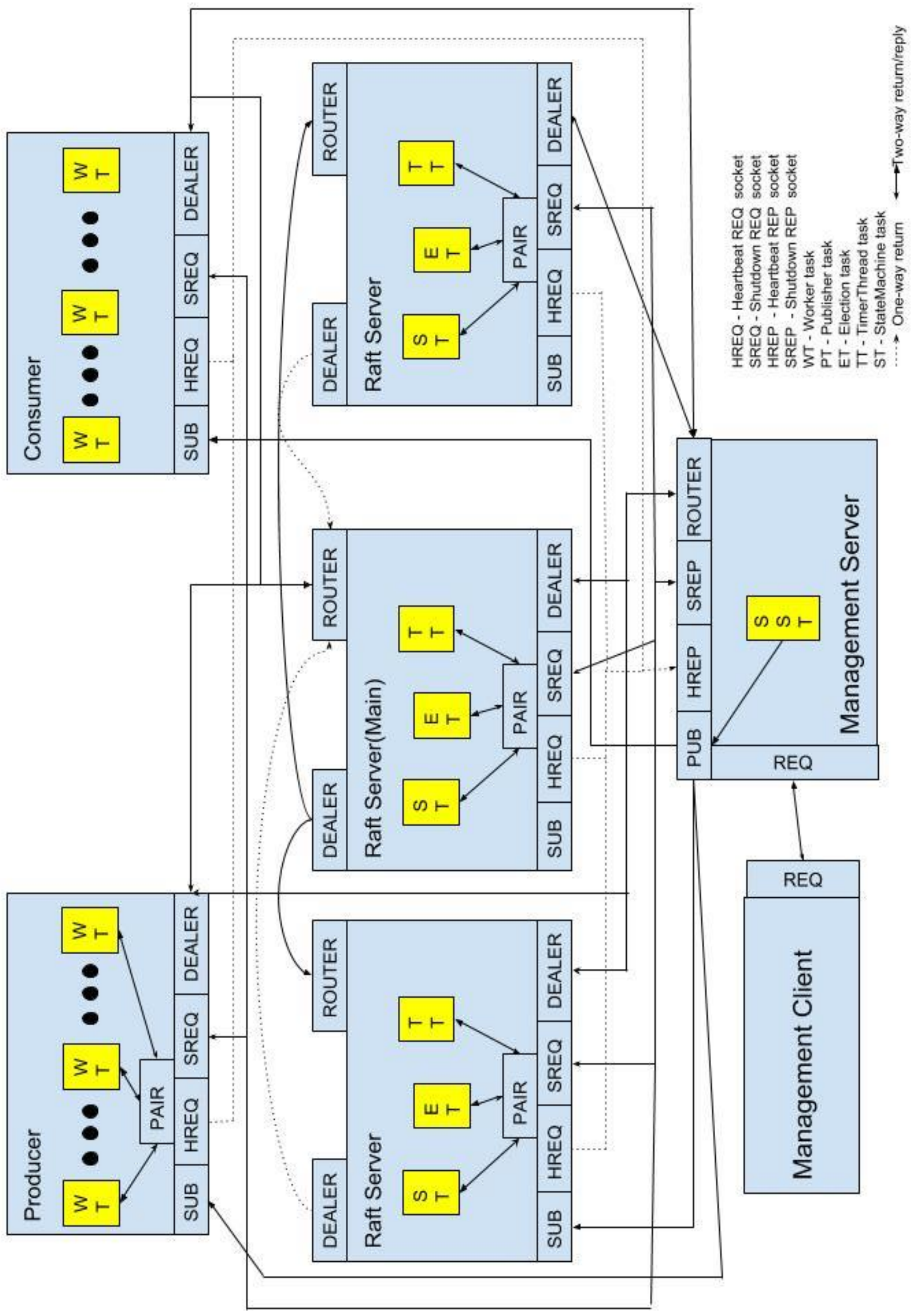


Figure 5.4. The RAFT socket architecture of the simulation

The RAFT server containers have three processes:

- a parent process which is the *ET*(election task) process which also houses the *ZMQ Poller,*

- a child process which is the *TT*(timerThread task). The *TT*(timerThread task) signals the ET when to start an election by sending a timeout message to it, and for the main RAFT server, signals when to send messages from the main RAFT servers log to the other servers for replication.

- a child process, which is the *ST*(StateMachine Task). The *ST*(StateMachine Task) represents the statemachine which is composed of a filethread. The *ST*(StateMachine Task) handles writing of the different RAFT persistent properties[22] and the committed entries on the RAFT log to the filesystem.

5.3.2 Client/server configuration

A cluster node in this configuration is a RAFT server as show in Figure 5.4. The client/consumer and server/producer are entities that request entries from the RAFT server (leader)'s log as in the case of the client/consumer or push entries to the RAFT server (leader)'s log as in the case of the server/producer. The consumer processes the received messages.

Upon start, the consumer and producer receive a set of cluster nodes from the management server. They initialize a set of DEALER sockets (*negotiation sockets*) whose number is equal to the number of cluster nodes in the system. These negotiation sockets each have a pair of addressable names: a private node name, for communication among themselves and a public node name for communication to the management server in Figure 5.4. They ping these cluster nodes using the DEALER sockets, in a round-robin fashion to find out which node is the actual leader. This “pinging” is shown as code in Figure 5.5. The consumer begins to request for entities from the leader, after connecting to the leader. The producer begins to push entities to the leader, after connecting to the leader.

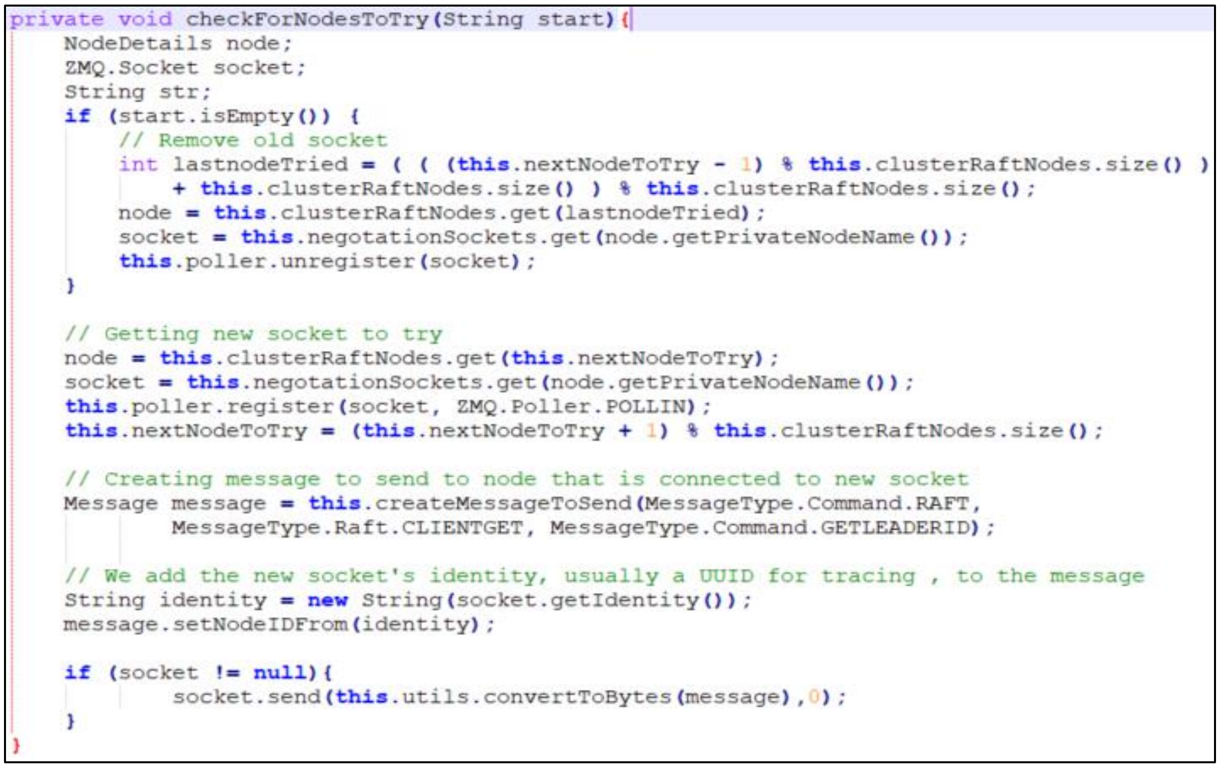
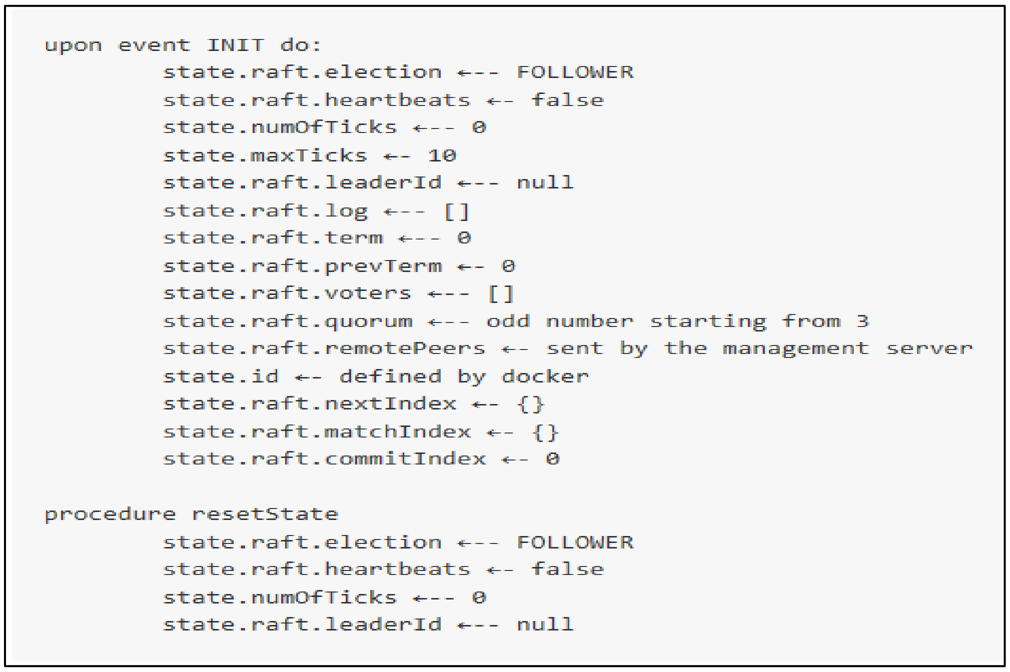

```java
private void checkForNodesToTry(String start){
    NodeDetails node;
    ZMQ.Socket socket;
    String str;
    if (start.isEmpty()) {
        // Remove old socket
        int lastnodeTried = ( ( (this.nextNodeToTry - 1) % this.clusterRaftNodes.size() )
            + this.clusterRaftNodes.size() ) % this.clusterRaftNodes.size();
        node = this.clusterRaftNodes.get(lastnodeTried);
        socket = this.negotationSockets.get(node.getPrivateNodeName());
        this.poller.unregister(socket);
    }

    // Getting new socket to try
    node = this.clusterRaftNodes.get(this.nextNodeToTry);
    socket = this.negotationSockets.get(node.getPrivateNodeName());
    this.poller.register(socket, ZMQ.Poller.POLLIN);
    this.nextNodeToTry = (this.nextNodeToTry + 1) % this.clusterRaftNodes.size();

    // Creating message to send to node that is connected to new socket
    Message message = this.createMessageToSend(MessageType.Command.RAFT,
            MessageType.Raft.CLIENTGET, MessageType.Command.GETLEADERID);

    // We add the new socket's identity, usually a UUID for tracing , to the message
    String identity = new String(socket.getIdentity());
    message.setNodeIDFrom(identity);

    if (socket != null){
            socket.send(this.utils.convertToBytes(message),0);
    }
}
```

Figure 5.5. The node choosing algorithm

5.3.3 General algorithm

The general algorithm used by the RAFT server is shown below. The general algorithm is an event loop that triggers a particular socket on inbound events. The three major events in the system are INIT, TIMEOUT, and RAFT. The INIT event comes from the management server. The INIT event causes the RAFT server to initialize all the variables to be used in the RAFT algorithm. Those variables and their initial values are shown in Figure 5.6:



```
upon event INIT do:
        state.raft.election ←-- FOLLOWER
        state.raft.heartbeats ←- false
        state.numOfTicks ←-- 0
        state.maxTicks ←- 10
        state.raft.leaderId ←-- null
        state.raft.log ←-- []
        state.raft.term ←-- 0
        state.raft.prevTerm ←- 0
        state.raft.voters ←-- []
        state.raft.quorum ←-- odd number starting from 3
        state.raft.remotePeers ←- sent by the management server
        state.id ←- defined by docker
        state.raft.nextIndex ←- {}
        state.raft.matchIndex ←- {}
        state.raft.commitIndex ←- 0

procedure resetState
        state.raft.election ←-- FOLLOWER
        state.raft.heartbeats ←- false
        state.numOfTicks ←-- 0
        state.raft.leaderId ←-- null
```

Figure 5.6. Initial values of some RAFT state variables and procedure *resetState*

The internal state of the server is denoted by *state*. The internal RAFT state is denoted by *state.raft*. The internal state of the RAFT messages sent between the peers is denoted by *message.raft*.

The TIMEOUT event is for the leader election algorithm discussed in previous chapters. The TIMEOUT event comes from the *TT* in Figure 5.4. The timeout indicates that an election can happen and the RAFT server can choose to act on it or disregard it. The RAFT event handles replication of the messages on the RAFT server's (leader) log. The RAFT event comes from the other RAFT servers in the cluster. Sections 5.3.3.1, 5.3.3.2, 5.3.3.3 are high level descriptions of the processes that occur on receipt of an INIT, TIMEOUT, RAFT events respectively.

5.3.3.1 INIT event

(a) The RAFT server waits upon INIT to receive a list of remote peers from the management server on its *DEALER* socket.

(b) Once it does (a), it starts up *TT* by sending a Timeout message to *TT*. *TT* then responds, after a certain randomized period of time known as an initial election timeout *ieT*, with a timeout message. *TT* runs for the entire lifetime of the RAFT server.

(c) On the completion of (b), the election algorithm described in the next section is run.

(d) On the termination of the election algorithm, the leader sends a heartbeat message to *TT*. *TT* then responds, immediately with a heartbeat message and goes to sleep for a certain timeout period, heartbeat timeout, *hT*.

(e) After (d), the replication algorithm then starts.

5.3.3.2 TIMEOUT event

(a) Upon receipt of the timeout message from *TT*, the RAFT server checks if it is a follower and if it has a leader.

(b) If the RAFT server is a follower and it does not have a leader, the RAFT server changes its *state.raft.election* to CANDIDATE. The RAFT server also executes the following steps:

(I) The RAFT server then resets its election timeout by sending a timeout message to *TT*. *TT* generates a new election timeout number, *eN* and responds after *eN* has expired.

(II) The RAFT server builds a *requestvote* message which contains these properties of the RAFT server: *state.id*, *state.raft.term*, *lastLogIndex*, *lastLogTerm*.

(III) The RAFT server then broadcasts it to all of its remote peers.

(c) If the RAFT server is not a follower, it disregards the timeout message.

(d) If the RAFT server is a follower and it does have a leader, it increments a counter, *state.numOfTicks* and checks to see if it is equal to another counter, *state.maxTicks*. If it is, it resets its state. Resetting its state makes the RAFT server ready to trigger an new election upon the receipt of the next timeout message. This *state.numOfTicks* is used to wait for the leader that could be unresponsive, that is, a heartbeat message has not been received by the RAFT server recently. The *state.maxTicks* is the upper limit to “time” to wait for the leader, before the RAFT server triggers a new elections. These counters are used to avoid situations where the RAFT server would wait unnecessarily for an unresponsive leader before the RAFT server is triggered by *TT*.

(e) The RAFT server checks if it is a leader. This also means that the RAFT server knows the leader in the cluster. If the RAFT server is a leader, the RAFT server broadcasts an *appendEntries* messages. If the RAFT server does not know the leader of the network, the RAFT server changes its *state.raft.election* to FOLLOWER.

5.3.3.3 RAFT event

The RAFT event comes with certain messages, *requestvote, requestvote_rep, appendEntries* and *appendEntries_rep* messages. Explanations of the algorithms that are run for each message are given below:

- Upon receipt of a *requestvote* message from a remote peer, the RAFT server checks for the following conditions:
- If the message term is greater than the RAFT server’s term, the RAFT server changes its state to that of a follower and sets its term to that of the message.
- If the message term is less than its term, the RAFT server sends a *requestvote_rep* message with a NACK back to the remote peer requesting a vote.
- If the message term is the same as its term and its log entries are the same as that of the message, the RAFT server sends back a *requestvote_rep* with an ACK message.

- If none of the above condition hold, the RAFT server sends a *requestvote_rep* message with a NACK back to the remote peer requesting a vote.
- Upon receipt of a *requestvote_rep* message from a remote peer, the RAFT server checks to see if the *requestvote_rep* message comes with an ACK. If the *requestvote_rep* message does, the RAFT server increments number of nodes that voted for it, *state.raft.voters*. If the *state.raft.voters* value is equal to the assigned quorum, *state.raft.quorum*:
  (a) the RAFT server changes its *state.raft.election* to LEADER
  (b) the RAFT server broadcasts a *setLeaderId* message which contains its id to all of its remote peers.
  (c) the RAFT server assigns for each remote peer, a match index of 0 and a next index of 1. The purpose of the match index is to indicate the highest entry replicated on a remote peer and the next index, the index of the next log entry of the RAFT server to send to that remote peer.
  (d) the RAFT server broadcasts a message with the following properties (*id, term, prevLogIndex, prevLogTerm, entries, commitIndex*)
- Upon receipt of an *appendEntries* message from a remote peer, the RAFT server runs through a series of steps:
  (a) The RAFT server checks if it is a leader or candidate. If it is either, it changes its *state.raft.election* to FOLLOWER. In the case if it is a leader, it disregards (b).
  (b) The RAFT server checks if the RAFT server has the same previous term as that of the message.

  - If the RAFT server does, the RAFT server checks to see if the message contains any new entries. If the message does not, the RAFT server sends back a heartbeat to the remote peer. If the message does, the RAFT server adds the new entries to its log. The RAFT server also responds with an *appendEntries_rep* message bundled with an ACK. If the commit index of the message is greater than the RAFT server's commit index, the RAFT server's commit index, *state.raft.commitIndex* becomes the min of the message's last log index and the message's commit index.
  - If the RAFT server does not, it responds with an *appendEntries_rep* message bundled with a NACK.

- Upon receipt of a *requestvote_rep* message from a remote peer, the RAFT server runs through a series of steps:

  (a) The RAFT server checks to see if the term in the message and in the RAFT server's term are equal.

  (b) If the term in the message is less than that of the RAFT server, the RAFT server disregards the message.

  (c) If the term in the message is greater than that of the RAFT server, the RAFT server changes its *state.raft.election* to FOLLOWER.

  (d) If the term in the message matches that of the RAFT server, the RAFT server checks for the message(responses) bundled with the *requestvote_rep* message:

  - If the response is an ACK, the RAFT server assigns the match index for the remote peer to be message's last log index. The RAFT server assigns the commit index to be the max of all the commit indexes of all of the remote peers. The RAFT server then assigns the next index for the remote peer to the min of the RAFT server's last log index and the message's last log index.

- If the response is a NACK, the RAFT server assigns the next index for the remote peer to be the conflicting index sent in the message.
- If the response is a heartbeat, disregard.

5.4 CRDT

5.4.1 Overall system architecture

The overall socket/docker/system architecture is shown in Figure 5.7. Each box contains a *ZMQ Poller* which uses *select()* and *poll()* internally to deque messages received by the sockets to be made available for use in the program. Every socket in the box is registered to a *ZMQ Poller*. From here on out, every time the word *container* with respect to Figure 5.7 is used, it represents a box in that image.

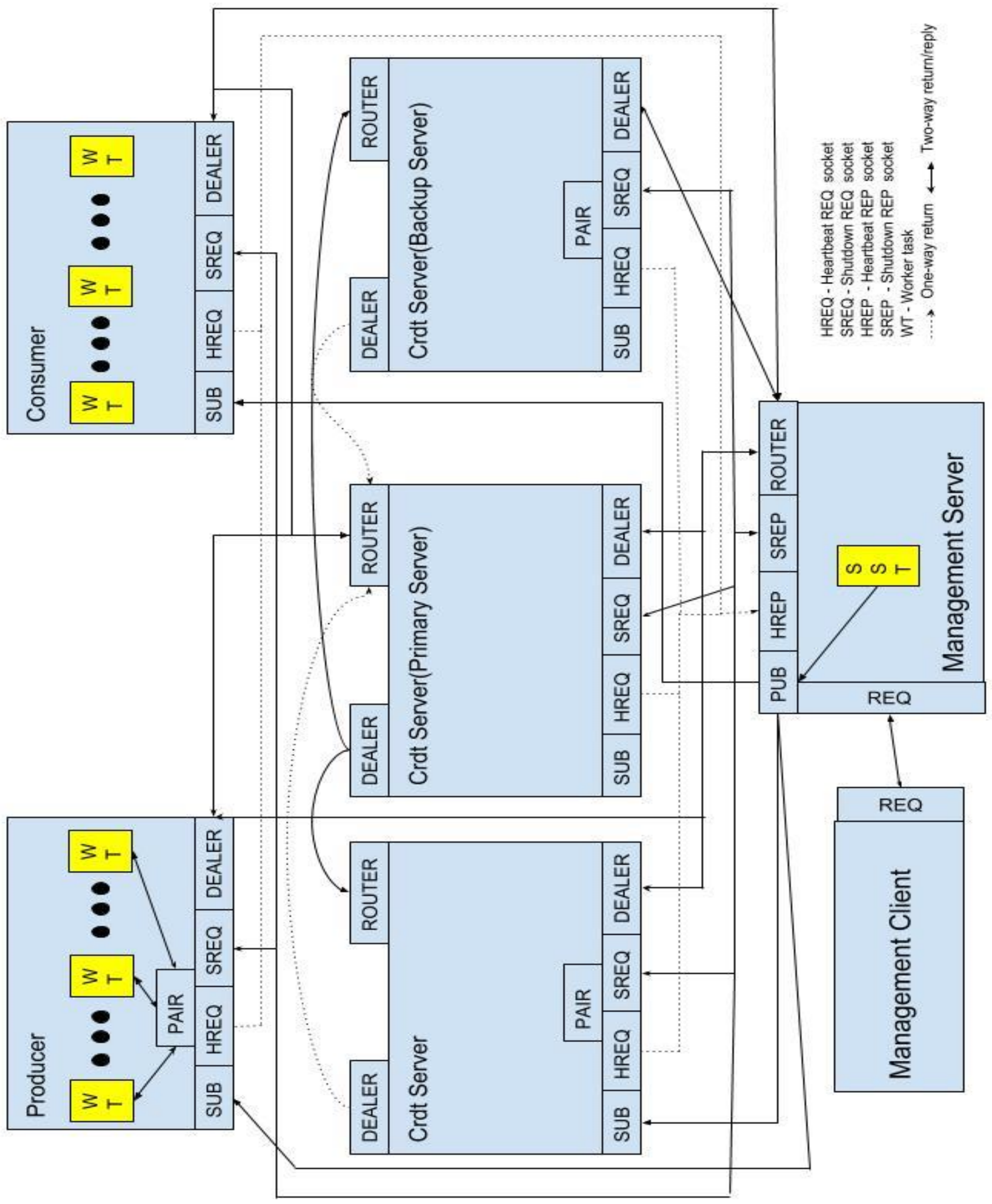


Figure 5.7. The CRDT socket architecture of the simulation

5.4.2 Client/server configuration

In the CRDT configuration, the consumer and producer role are similar to the description in Section 5.3.2. The CRDT configuration uses a binary star pattern [20] for reliability.

In the binary star pattern [20], there are two main servers that all the peer nodes connect to. One of the pair of the servers is the primary server and active server, that is, it is the server that the outside world (consumer and producer) connect to. The other server in the pair is the backup and inactive server. The client and producers do not have a direct connection to the any other nodes in the network apart from the above pair. If the active server were to go down, the outside world and all the peers in the cluster will automatically connect to the backup server. The backup server notifies the management server from Figure 5.7 and proceeds to assume the role of the active and primary server. The management server then selects a new backup and inactive server from the remaining peers.

The peer nodes that are not active or backup are connected to each other and do not go through the server to replicate messages. In this section and in Section 5.4.3, they are referred to as the *ordinary servers.* The backup server does not participate in replication until it is made active and primary. Each peer has a log where the messages are stored and the log is a state-based CRDT, a *map*. The operation of the map is discussed in Section 4.1.2.

5.4.3 General algorithm

The general algorithm used by the CRDT server is shown below. The general algorithm is an event loop that triggers a particular socket on inbound events. The two major events in the system are CONNECTION and CRDT. The CONNECTION event comes from the management server and from the outside world (consumer and producer). The CONNECTION event from the management server determines if the CRDT server is chosen as active or inactive. The CONNECTION event from the outside world is used to give the calling node the addresses of the active and backup servers. The CRDT event is used for replication of messages from one peer node to another. Sections 5.4.3.1, 5.4.3.2 are high level descriptions of the processes that occur on receipt of CONNECTION and CRDT events respectively.

5.4.3.1 CONNECTION event

- Upon receipt of a connection event from the management server, the CRDT server checks to see if the management server designates the CRDT server as a primary and active server or as a backup and inactive server. Whatever the outcome, the CRDT server sends back an ACK to the management server that it changed its state. The CRDT server then broadcasts a *setPrimary* or *setBackUp* message to every peer in the cluster.
- Upon receipt of a connection event from the outside world (consumer and producer), the CRDT server checks to see it has the address of both the active and the backup servers. If the CRDT server does, it sends back a *setServers* message with the addresses. If the CRDT server does not, it sends a *setServers* message with a NACK.

5.4.3.2 CRDT event

The CRDT event comes with certain messages: *serverPost, update, clientGet, serverGet, mergeState, mergeState_rep, serverGet_rep, clientGet_rep, removeValu*e messages. Explanations of the algorithms that are run for each message are given below:

- Upon receipt of a *serverPost* message from the producer, the CRDT server checks to see if it is an ordinary server. If it is not, it disregards the message. If the CRDT server is an ordinary server, it checks to see if the message is a delta or not.
  - If the message is a delta, the CRDT server updates its event log and broadcasts the delta as an update message to all the other ordinary servers.

  - If the message is not a delta, the CRDT server sends its entire log as an update message to all the other ordinary servers.
- Upon receipt of an *update* message from an ordinary server, the CRDT server checks to see if it is an ordinary server. If it is not, it disregards the message. If the CRDT server is an ordinary server, it checks to see if the message is a delta or not.
  - If the message is a delta, the CRDT server updates its event log with the delta.
  - If the message is not a delta, the CRDT server updates its event log with the message payload.
- Upon receipt of a *clientGet* message from the consumer, the CRDT server checks to see if it is an ordinary server. If it is, it disregards the message. If the CRDT server is a primary server, it broadcast a *serverGet* message to all the ordinary servers.
- Upon receipt of a *serverGet* message from the primary server, the CRDT server checks to see if it is an ordinary server. If it is not, it disregards the message. If the CRDT server is an ordinary server, it broadcasts a *mergeState* message to all the other ordinary peers.
- Upon receipt of a *mergeState* message from an ordinary server, the CRDT server checks to see if it is an ordinary server. If it is not, it disregards the message. If the CRDT server is an ordinary server, it does a merge, according to the merge operation, detailed in Chapter 5. The CRDT server then appends its merged state to its event log. The CRDT server then sends a *mergeState_re*p back to the CRDT server where the *mergeState* message originated from.
- Upon receipt of a *mergeState_rep*, the CRDT server gets its updated state and sends it to the primary server as part of a *serverGet_rep* message.

- Upon receipt of a *serverGet_rep* message, the CRDT server checks to see if it is an ordinary server. If it is, it disregards the message. If the CRDT server is a primary server, the CRDT server increments a counter, *mergedStates*. Once the counter is equal or greater than the number of cluster nodes minus 2, the CRDT server merges the state in the *serverGet_*message with its own event log. The CRDT server then sends the latest value of the state for the value the consumer is requesting, as part of a *clientGet_rep* message. The CRDT server then sends that latest value as part of a *removeValue* message.
- Upon receipt of a *removeValue* message, the CRDT server checks to see if it is an ordinary server. If it is not, it disregards the message. If the CRDT server is an ordinary server, it does a remove operation as described in Chapter 4 with the payload of the *removeValue* message. This is done to prevent the event logs of the ordinary server from growing too large with all the frequent merges of state.

Chapter 6: Results

In this chapter, a comparison of the two implementations of CRDT and RAFT as described in Chapter 5 was performed. Two metrics, latency and throughput were used as the basis for the comparison. In the subsequent section, the simulation environment will be described. Next, a definition of the metrics used will be explained. Finally, results in the form of graphs will be shown and explained.

6.1 Simulation Environment

The simulation was run on a laptop, running Ubuntu OS 16.04.3. The processor is an Intel Core i7-4500U (4th Gen) with 1.8 Ghz clock speed. The memory is 8GB DDR3 RAM and the hard disk size is 1TB. The simulation was run with the following software, Docker 18.0.1 CE, Java 8, jzmq-4.0.1.

6.2 Metrics

The metrics used for comparison are latency and throughput. Latency is the time it takes for a single message to go from the producer to the consumer and is measured in seconds. Throughput is the inverse of latency and is the number of messages that could be delivered in one second at the consumer from the producer, after passing through the message queue. This latency and throughput metric are averages, as a baseline for the message queue is desired. The computation of the latency and throughput metric follow the model for computing latency and throughput in [23] and is described below.

A particular number of messages (N = 2000) are tagged with a tag Id, t_d. t_d is necessary for the consumer to identify messages with t_d and record their timestamps as part of the simulation. Timestamps of messages that have been previously received by the consumer are not recorded. This is because we want to measure *replication* latency, not *message* latency. Based on the description of the RAFT and CRDT implementations in Chapter 5, the consumer can receive the same message multiple times.

Each message *w*, generates a timestamp *t* upon initialization. This timestamp correlates to O_n in [23]. This same message is what is retrieved for the RAFT implementation. The CRDT implementation does this differently. Since a new message *m* is always generated from a message *q* during merge operations, a merge operation copies timestamp *t* into *q*. This makes it equivalent to what the RAFT implementation does.

The latency and throughput computations are done on the consumer, because with that, we can be sure that a particular message *m* was sent from the producer, through the queue and received by the consumer. Once the consumer receives message *m* that has tag id *td* and has not yet received the same message before, it records the timestamp t_m as the received timestamp. This correlates to I_n in [23].

The latency L_n, then becomes the difference between I_n and O_n. The average latency is calculated as L divided by the number of messages tagged with t_d. The throughput, T is computed as the number of messages tagged with t_d, divided by the difference between time the last message was received and the time the first message was received. To measure the latency/ throughput, the concept of the simulation run is used.

A *simulation run* is pushing 2000 messages through the queue from the producer to the consumer. This simulation run is repeated 20 times and average values were computed. The average latency and throughput on the graphs, shown in Figures 16-23 is the average of the repeated simulations. The average time for one simulation run is 1.5 hrs.

We test the throughput and latency against the number of replicas and also against message sizes. These are two interesting parameters to judge the message queue performance on, due to two reasons:

- Size of the payload: This is to determine how the latency is affected under different message sizes. In these simulations, we use a small message size (128 bytes) and a large message size (512 bytes).

- No of replicas: This is to determine what effect an increasing number of replicas will have on latency. This increase, in a normal application might be due to fault tolerance and compensate for node failures. In these simulations, we test the message queue with 3, 5, 7 replicas.

6.3 Numerical Results

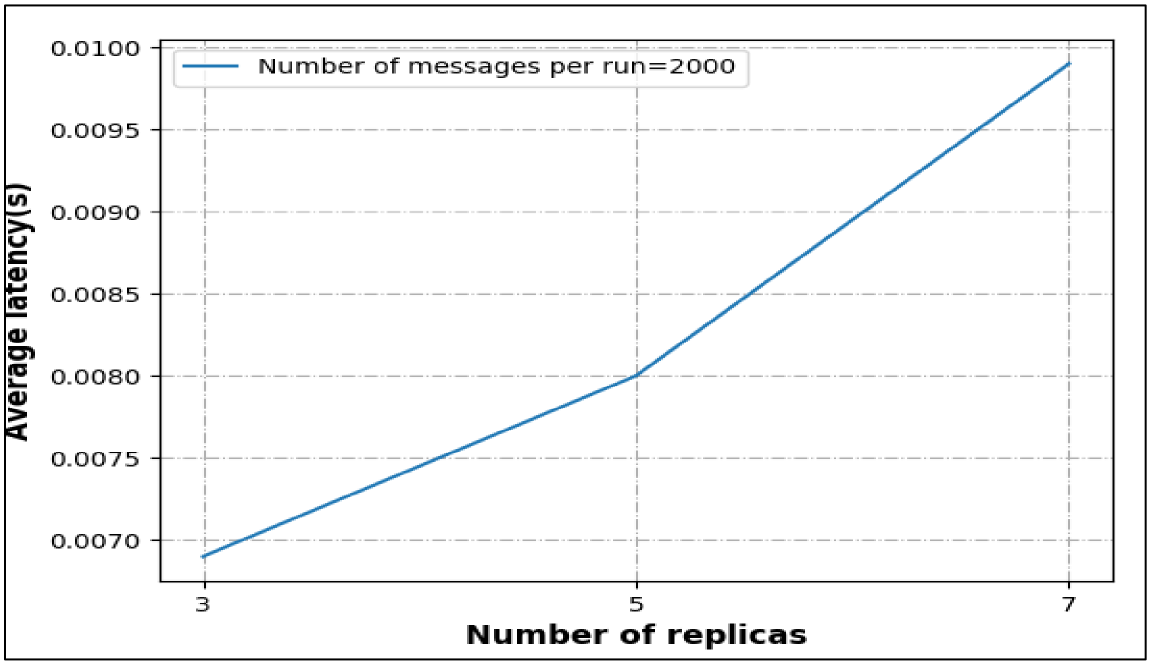


Figure 6.1. Average latency for message size 128 bytes (RAFT)

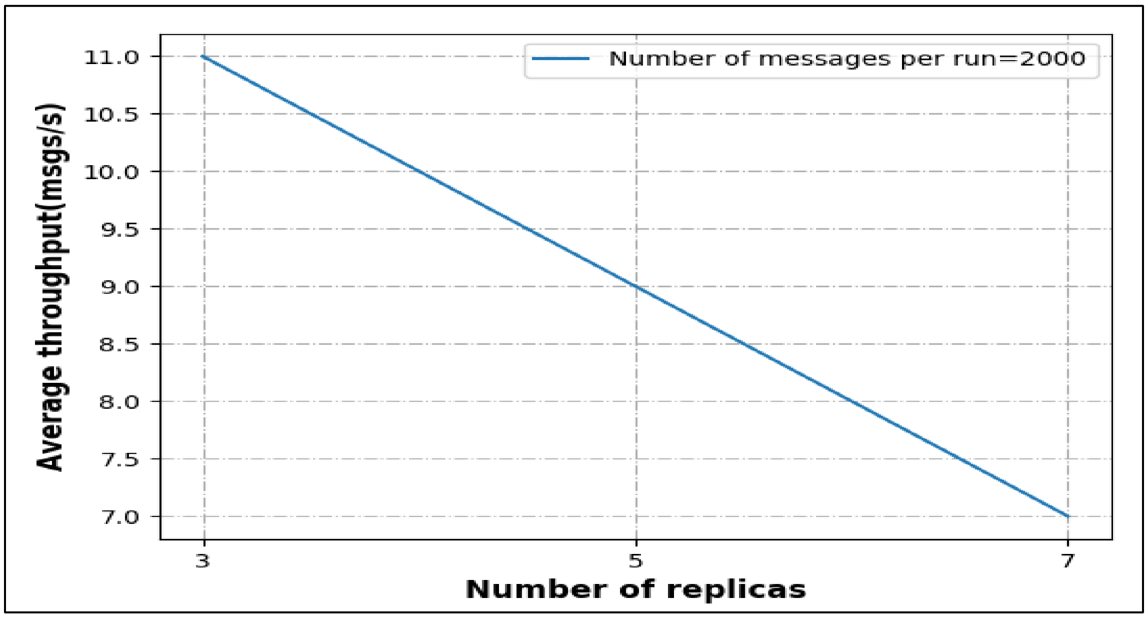


Figure 6.2. Average throughput for message size 128 bytes (RAFT)

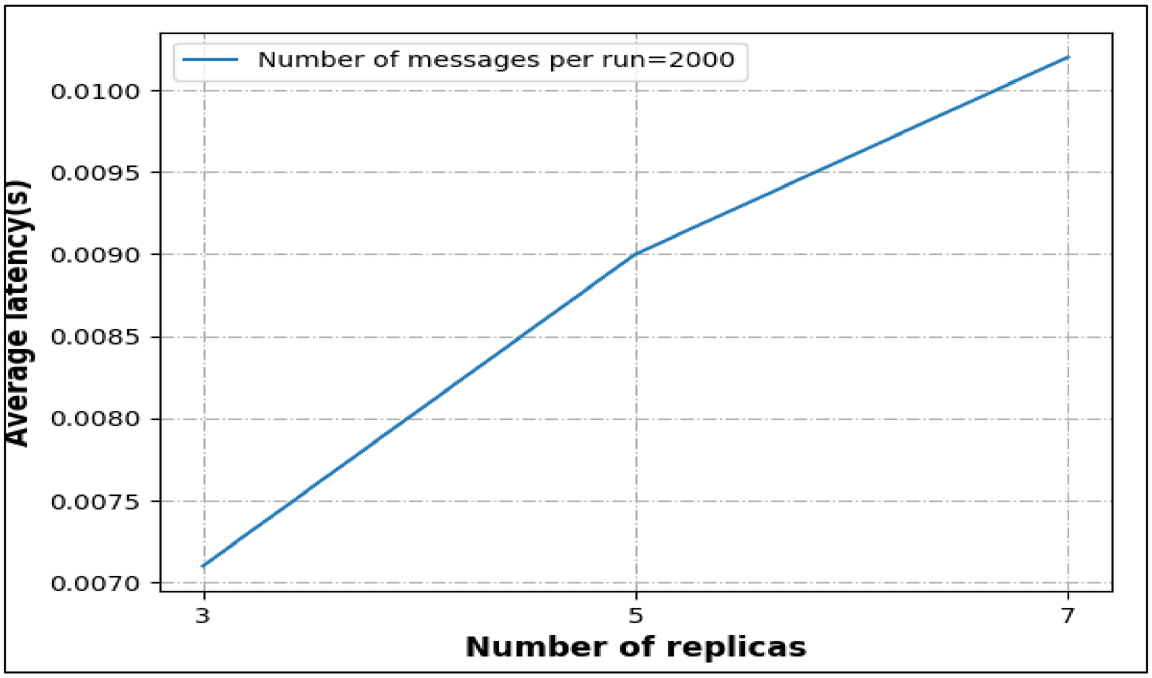


Figure 6.3. Average latency for message size 512 bytes (RAFT)

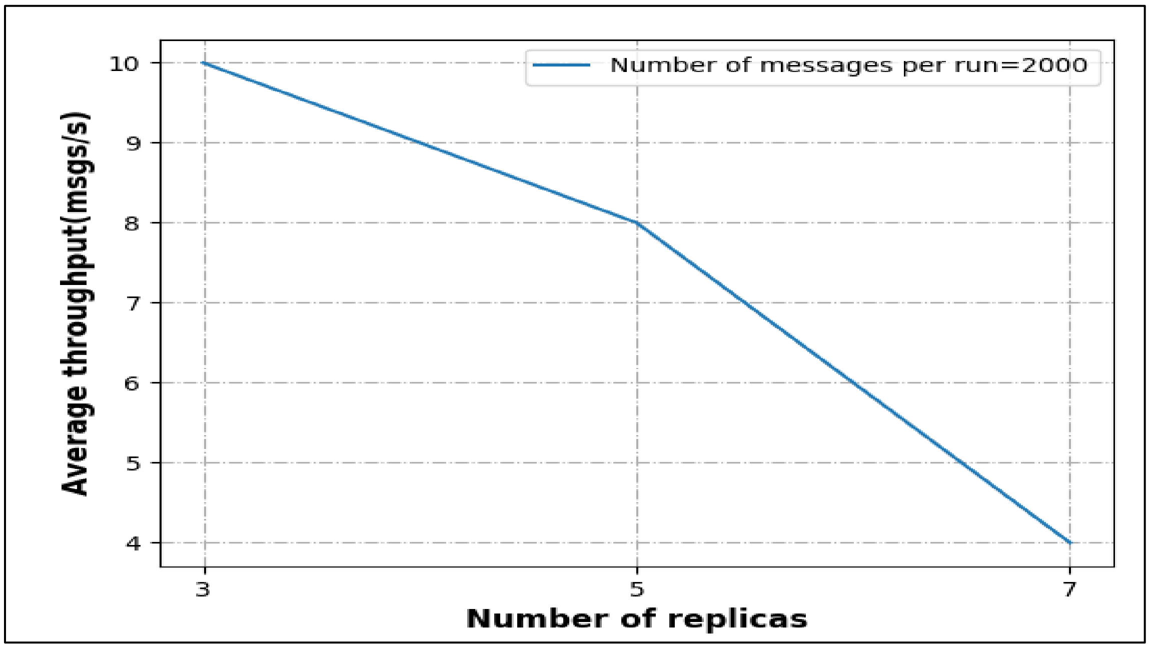


Figure 6.4. Average throughput for message size 512 bytes (RAFT)

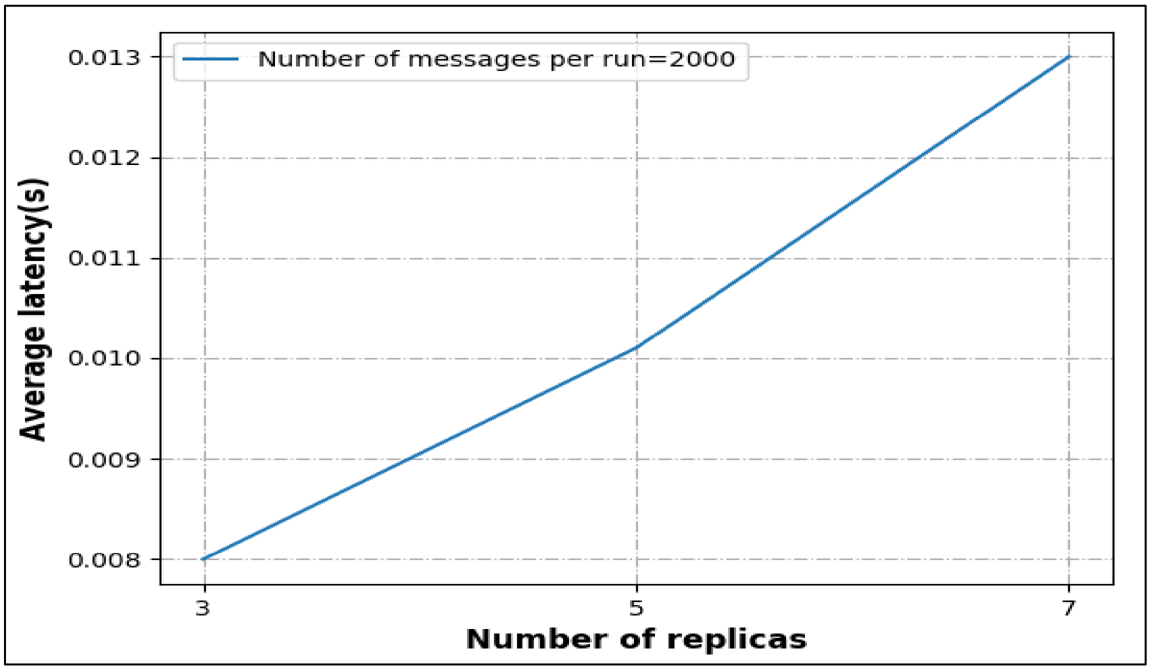


Figure 6.5. Average latency for message size 128 bytes (CRDT)

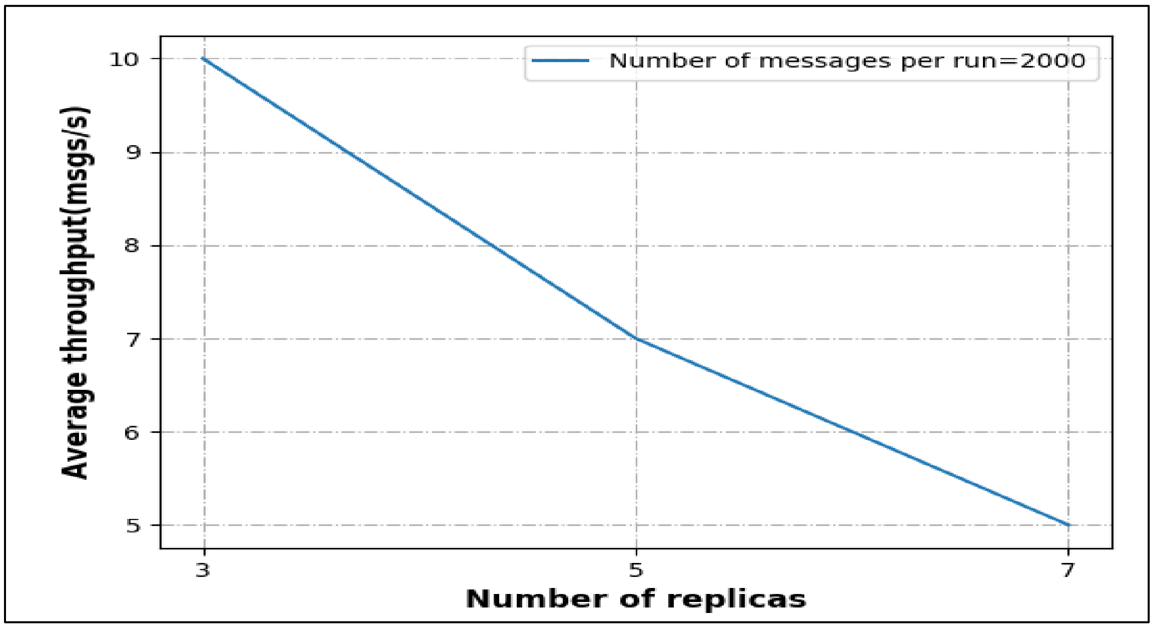


Figure 6.6. Average throughput for message size 128 bytes (CRDT)

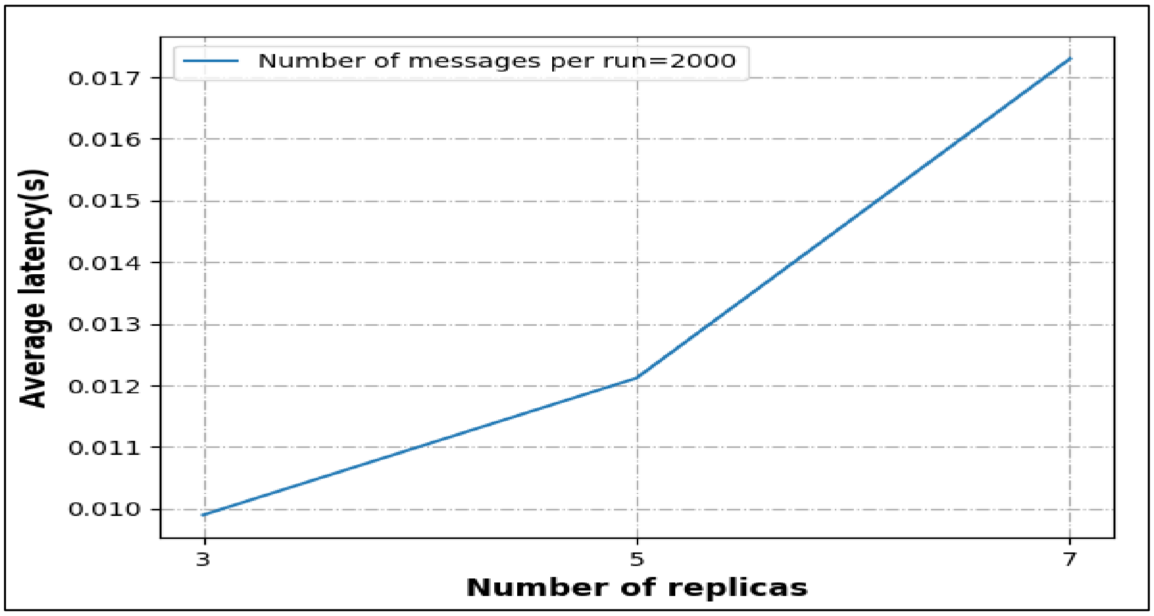


Figure 6.7. Average latency for message size 512 bytes (CRDT)

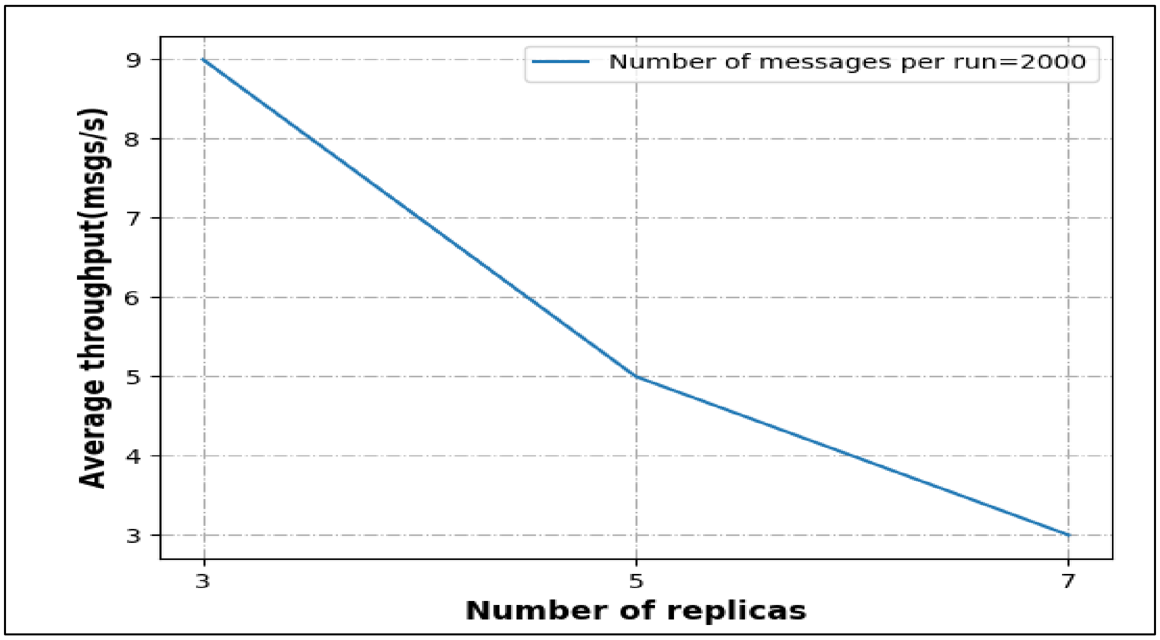


Figure 6.8. Average throughput for message size 512 bytes (CRDT)

Figures 6.1-6.4 is the graphs for the RAFT implementation. Figures 6.5-6.8 are the graphs for the CRDT implementation. For both implementations, as the number of replicas increase, a increase in latency is observed. This increase can be explained by the fact that the more nodes would result in more messages which would increase the time to indicate that a particular message has been replicated. The increase is sharper for CRDT than it was for RAFT due to the nature of the implementation requiring more communication for replication than RAFT. A drop is observed for throughput. This means that latency is inversely proportional to throughput, that means that higher values for latencies will result lower values for throughput. The message sizes also affected both implementations as both systems seemed to perform better for the smaller message (128 bytes) than for the larger message (512 bytes).

Chapter 7: Conclusion and Future Work

From the experiments carried out in this paper, the RAFT [2] protocol is shown to have a better message throughput and latency than that of the CRDT [3] protocol. From the experiments carried out in the paper, both systems spend quite a bit of time replicating their messages, given that they both process less than 20 messages per second.

In this paper, we discussed and implemented the RAFT protocol (leader election and replication). We also discussed the CRDT and the data types that can be derived from CRDT. We implemented the map and LWW set data types from the CRDT as CmRDTs. We implemented these protocols with Docker to simulate a distributed system, which is a novel implementation to date.

Some possible future work are as follows:

- Adding network instability to test the system under failures.
- Implementing the CRDT protocol for operation-based replication, a CvRDT which could greatly reduce the number of messages sent between nodes.
- Implement the extended parts of the RAFT protocol such as log compaction [22], and snapshotting [22].
- Running the CRDT and RAFT implementations on Amazon AWS EC2 [24] to simulate conventional distributed load.
- Tune the systems to see if the throughputs and latencies could be improved upon.

References

[1] E. A. Brewer, “Towards robust distributed systems,” *PODC,* vol. 7, 2000.

[2] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” *USENIX Annual Technical Conference (USENIX ATC 14).* 2014.

[3] M. Shapiro *et al.*, “Conflict-free replicated data types,” in *Symposium on Self-Stabilizing Systems.* Springer Berlin Heidelberg, 2011.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of convergent and commutative replicated data types,” Ph.D. Dissertation Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[5] M, Sustrik. “ZeroMQ” [Online]. Available: http://www.aosabook.org/en/zeromq.html. [Accessed: October 10, 2016].

[6] L. Lamport, “The part-time parliament,” in *ACM Transactions on Computer Systems (TOCS),* vol. 16, no. 2, pp. 133-169, 1998.

[7] L. Lamport, “Paxos made simple,” in *ACM Sigact News*, vol. 32, no. 4, 2001, pp. 18-25.

[8] CoreOS Fest 2015, “An introduction to raft,” *Youtube*, May 27, 2015. [Video File]. Available: https://www.youtube.com/watch?v=6bBggO6KN_k. [Accessed: October 17, 2017].

[9] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990, pp. 299-319.

[10] D. Dziuma *et al.*, “Survey on consistency conditions,” *Forth-Ics Tr.,* vol. 439, 2013, pp.1-26.

[11] S. V. Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *IEEE Computer,* vol. 29, no. 12, pp. 66-76, December 1996.

[12] P. Alvaro *et al.*, “Consistency analysis in bloom: A CALM and collected approach,” *CIDR*, 2011, pp. 249-250.

[13] D. S. Parker *et al.*, “Detection of mutual inconsistency in distributed systems,” *IEEE Transactions on Software Engineering*, vol. 3, pp. 240-247, 1983.

[14] J. S. Almeida, S. A. Paulo, and B. Carlos, “Bounded version vectors,” in *International Symposium on Distributed Computing.* Berlin: Springer Heidelberg, 2004, pp. 102-116.

[15] G2Crowd, “Digital trends: microservices,” *g2crowd.com* [Online]. Available: https://blog.g2crowd.com/blog/trends/digital-platforms/2018-dp/microservices/ [Accessed Oct. 10, 2016].

[16] R. Brown *et al.*, “Riak DT map: A composable, convergent replicated dictionary," in *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency,* ACM, p. 1, 2014.

[17] D. Merkel, “Docker: Lightweight Linux containers for consistent development and deployment,” in *Linux Journal*, vol. 201, no. 239, 2014, p. 2.

[18] Smart Home Beginner, “What is Docker: Docker vs VirtualBox, Home server with Docker,” *smarthomebeginner.com* [Online]. Available: https://www.smarthomebeginner.com/what-is-docker-docker-vs-virtualbox/ [Accessed: May 19, 2018].

[19] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs containerization to support paas” in *2014 IEEE International Conference on Cloud Engineering (IC2E),* IEEE, pp. 610-614, March 2014.

[20] ØMQ. “ØMQ–the guide,” *zeromq.org* [Online]. Available: http://zguide.zeromq.org/page:all [Accessed June 18, 2018].

[21] ØMQ. “ØMQ–API,” *zeromq.org* [Online] Available: http://api.zeromq.org/4-2:zmq [Accessed June 25, 2018].

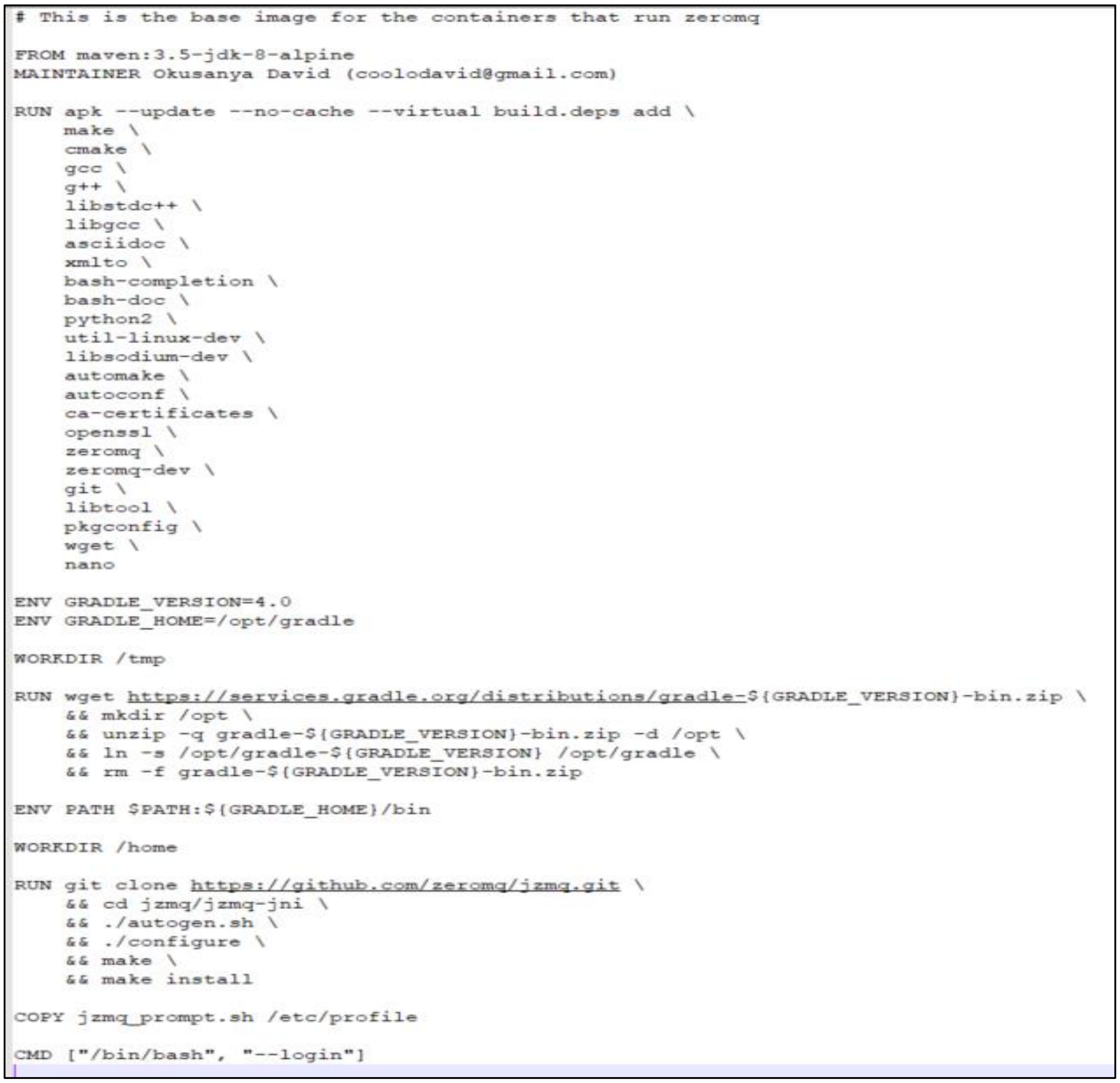
[22] H. Howard, *ARC: Analysis of Raft consensus*. University of Cambridge, Computer Laboratory, 2014.

[23] ØMQ, “Measuring messaging performance,” *zeromq.org* [Online] Available: http://zeromq.org/whitepapers:measuring-performance [Accessed: June 25, 2018].

[24] Amazon, “Amazon EC2,” *aws.amazon.com* [Online]. Available: https://aws.amazon.com/ec2/?c=1&pt=1 [Accessed: Oct 10, 2016].

Appendix A: Introduction to a Dockerfile

A Dockerfile[1] is a declarative way of instructing the *docker* program on how to build our *image*, which would be ran as a *container*. This declaration is done as a text file and is fed to a command, *docker build* which builds the *image*. The dockerfile for the base image of the simulation is shown below:



```
# This is the base image for the containers that run zeromq

FROM maven:3.5-jdk-8-alpine
MAINTAINER Okusanya David (coolodavid@gmail.com)

RUN apk --update --no-cache --virtual build.deps add \
    make \
    cmake \
    gcc \
    g++ \
    libstdc++ \
    libgcc \
    asciidoc \
    xmlto \
    bash-completion \
    bash-doc \
    python2 \
    util-linux-dev \
    libsodium-dev \
    automake \
    autoconf \
    ca-certificates \
    openssl \
    zeromq \
    zeromq-dev \
    git \
    libtool \
    pkgconfig \
    wget \
    nano

ENV GRADLE_VERSION=4.0
ENV GRADLE_HOME=/opt/gradle

WORKDIR /tmp

RUN wget https://services.gradle.org/distributions/gradle-${GRADLE_VERSION}-bin.zip \
    && mkdir /opt \
    && unzip -q gradle-${GRADLE_VERSION}-bin.zip -d /opt \
    && ln -s /opt/gradle-${GRADLE_VERSION} /opt/gradle \
    && rm -f gradle-${GRADLE_VERSION}-bin.zip

ENV PATH $PATH:${GRADLE_HOME}/bin

WORKDIR /home

RUN git clone https://github.com/zeromq/jzmq.git \
    && cd jzmq/jzmq-jni \
    && ./autogen.sh \
    && ./configure \
    && make \
    && make install

COPY jzmq_prompt.sh /etc/profile

CMD ["/bin/bash", "--login"]
```

Figure A.1. Dockerfile for the base image of the simulation

Each line of a Dockerfile is an instruction, which is run independently and serially. Docker images are layered filesystems [2]. When each line of a Dockerfile is run, it produces a layer which is added to the layer produced by the previous command. The summary of Figure A.1 is as follows:

(a) Build a layer from the official maven java-8-jdk-alpine image, using the *FROM* command.

(b) Use the *RUN* command to run a package installation using the package manager for the alpine OS and to also build the *jzmq* package.

(c) Setup the environment variables for *gradle* [3], which is used to install jzmq, with the *ENV* command

(d) Make the directory to run all the *.java* files with the *WORKDIR* command

(e) Copy a file *jzmq_prompt.sh* from the host directory into the */etc/profile* of the image, using the *COPY* command.

(f) Whenever a container is created from this image, provide the default login shell /bin/bash using the *CMD* command

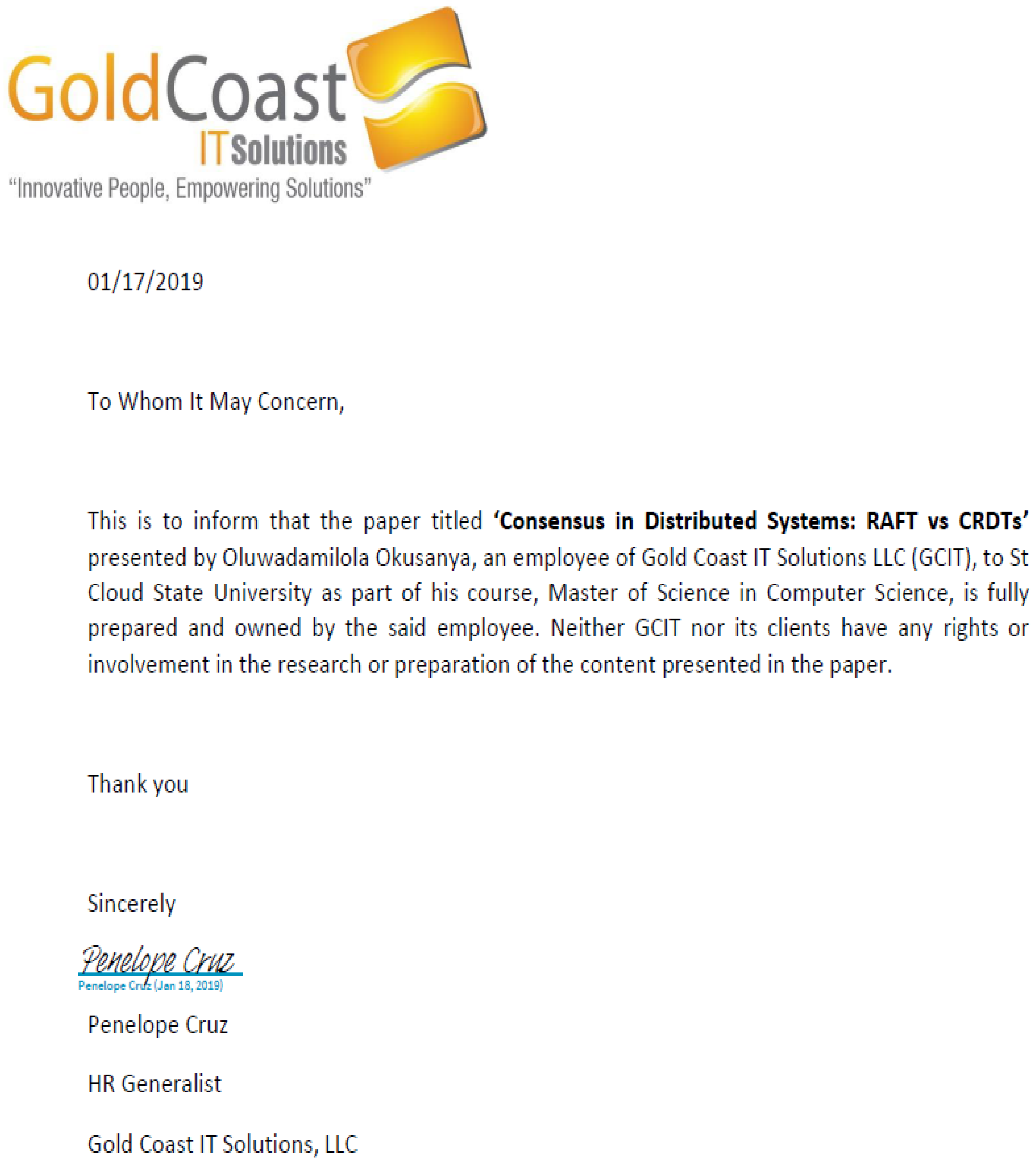
Appendix B: References

[1] Docker, “Dockerfile reference,” *docker.com* [Online]. Available: https://docs.docker.com/engine/reference/builder/ [Accessed: Oct 30, 2016].

[2] Usenix, “Layered file system,” *usenix.org* [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/freenix/full_papers/studenmund/studenmund_html/node7.html [Accessed: Oct 30, 2016].

[3] Gradle, “Gradle Build Tool,” *gradle.org* [Online]. Available: https://gradle.org/ [Accessed: Oct. 30,2016].

Appendix C: Signed IP Release Letter



GoldCoast IT Solutions

"Innovative People, Empowering Solutions"

01/17/2019

To Whom It May Concern,

This is to inform that the paper titled **'Consensus in Distributed Systems: RAFT vs CRDTs'** presented by Oluwadamilola Okusanya, an employee of Gold Coast IT Solutions LLC (GCIT), to St Cloud State University as part of his course, Master of Science in Computer Science, is fully prepared and owned by the said employee. Neither GCIT nor its clients have any rights or involvement in the research or preparation of the content presented in the paper.

Thank you

Sincerely

Penelope Cruz

Penelope Cruz (Jan 18, 2019)

Penelope Cruz

HR Generalist

Gold Coast IT Solutions, LLC