Project Report

Byzantine Fault Tolerant Raft

Team Members:
Ting-Chi Yeh(W1280548)
Shan He(W1287054)
Yujian Zhang (W1270711)
Yu-Cheng Lin(W1272075)

Under the guidance of:
Professor Ming-Hwa Wang

Department of Computer Science & Engineering
Santa Clara University, Santa Clara, CA

Acknowledgement

First and foremost, we would not work on this project and testing against our hypothesis if without this class, COEN 241 cloud computing. Furthermore, we would also give special thanks to Professor Wang for his rich knowledgement and teaching patiently in the class, from which we really learned a lot.

Table of content

Acknowledgement **2**

Table of content **3**

Introduction **5**

1.1 Objective 5

1.2 What is the problem 5

1.3 Why this is a project related to this class 5

1.4 Why other approach is no good 6

1.5 Why we think our approach is better 6

1.6 Statement of the problem 7

1.7 Area or scope of investigation 7

2. Theoretical Bases and Literature Review **8**

2.1 Theoretical background of the problem 8

2.2 Related research to solve the problem 9

2.3 Advantage/ disadvantage of those research 10

2.4 Our solution to solve this problem 11

2.5 Why our solution is better 12

3. Hypothesis/Goal **13**

4. Methodology **14**

4.1 How to generate/ collect input data 14

4.2 How to solve the problem 14

4.3 How to generate output 15

4.4 How to test against hypothesis 15

5. Implementation **16**

5.1 Code implementation 16

5.1.1 Code source 16

5.1.2 Language 16

5.1.3 Class Diagram 16

5.1.4 Key Components 16

Client Interface 16

Leader, Follower, Candidate 18
State Machine 19
Data Storage 21
Signed Message 22
TCP Communication & Timeout 24
5.2 Design document and flowchart 26
Leader Election 26
Normal RAFT 27
Byzantine RAFT 28
Flowchart: (Skeleton) 29

6. Data analysis and discussion 30
6.1 Output generation 30
6.2 Output analysis 30
6.3 Compare output against hypothesis 30
6.4 Discussion 30

7. Conclusion and recommendations 31
7.1 Summary and conclusions 31
7.2 Recommendations for future studies 31

8. Bibliography 33

Table of Figures

Figure 1 All the class implemented in the enhanced RAFT. Up right corner classes controls the state machine. Up right corner classes controls the signed communication. Center classes controls the RAFT character switching.

Figure 2 Leader election process in Original Raft.

Figure 3 In normal usage condition for RAFT.

Figure 4, Byzantine case for RAFT with Enhanced RAFT.

Figure 5 Flowchart for Enhanced RAFT.

Figure 6 The implementation for Enhanced RAFT at normal case.

Figure 7 The implementation for Enhanced RAFT at byzantine case.

1. Introduction

1.1 Objective

Attempt to introduce Byzantine fault tolerance into Raft[1].

1.2 What is the problem

In order to guarantee the system availability, the user request will be executed by all replicas. Raft assumes that nodes fail only in the way of crashing or delayed due to network congestion. All the messages transmitted between each two nodes are correct as expected and well-received. Agreement decision is concluded based on received messages. Therefore, fail symptom under this assumption is simply without response or request from that failed node. If leader is malfunctioning, followers will start leader election process due to no heartbeat message from the leader. Then, after another leader is elected, the system recovers from the failure and continues to work.

However, in reality, there are other fail symptoms that neglected by Raft, such as sending wrong messages. Due to processing requests incorrectly, malicious attacks, corrupting their local state, and/or producing incorrect or inconsistent output, faulty nodes will exhibit Byzantine behavior and consequently impact the correctness and availability [2][3]. Although the Raft is designed for educational and understandable, we would like to make it more practical by adopting some Byzantine fault solution [4] into Raft.

1.3 Why this is a project related to this class

Many cloud service are held in distributed environment. In distributed environment, many factors would decrease the availability, such as network issues and faulty hosts. The cloud service providers are responsible for making sure when some of their server host become dysfunctional, the service still works. One of the solution is to make replica into the system. The system needs to make sure consistency among the replicas, and this is where consensus algorithm take part. Raft is such a consensus algorithm. Before Raft, many use Paxos in their system. However, since Raft is designed to be more simple and understandable, it wins more and more attention to both academia and industry. Raft successfully solves faulty case that some servers are down, but leaves Byzantine problem unsolved, which potentially decrease its availability. Therefore, we try to make up part of this insufficiency by introducing approaches to solve the Byzantine problem.

1.4 Why other approach is no good

The goal of the Raft is to help understand the consensus algorithm and implementation in the real system for those who have trouble understanding Paxos [5][6][7], which is a very brilliant, but difficult algorithm to solve the consistency issue in the distributed system.

The Raft has made many assumptions to make the concept easier for learners. As in our interest, the Raft has assumed the node in the cluster would either work perfectly or not work at all. They did not consider the faulty node situation. Our goal is trying to make it more practical by supporting Byzantine fault tolerance. We would like to make the Raft still functional when some of the nodes among the cluster would not work as expected but still able to send message to others. In this case, each good node might need to distinguish from those fault message and need to have the same execution result while we might not care what would the fault node do.

1.5 Why we think our approach is better

People have invested a lot of engineering effort reducing the overhead, and amortizing the cost messages by sending these messages on top of other messages in order to solve Byzantine failure. In our approach, with signed message, we reduce the number of total communication to ensure Byzantine won't misleading the normal nodes.

1.6 Statement of the problem

We are interested in the situation when Byzantine node involved in the leader part in the Raft. In our project, we would like to solve the following situation:

1. when the system is functional and a leader already exists, one of the Byzantine node would keep sending request to become a new leader even when it become a leader later, if it does.
2. the Byzantine leader might instruct the followers to commit the replica even though there are not enough number of replicas have safely been stored in the durable place, hard disk for example.
3. Byzantine leader send different user request among followers.

1.7 Area or scope of investigation

Both the leader and the follower could be Byzantine node, but here, we only discuss the Byzantine leader.

2. Theoretical Bases and Literature Review

2.1 Theoretical background of the problem

Raft:

Raft is designed to build a more understandable consensus algorithm, but still contains practical potential usage, for education and some other situation to implement. Although the single-paxos is well defined and detailed explained, multi-Paxos is really hard to understand and due to the original Paxos did not consider this issue, it is really hard to implement a multi-Paxos into a really functional system, even Google's Chubby faced a lot of issues. Raft achieved a better understandability by using a leader/follower style. Only the leader could send replicating instruction to the follower and the cluster would eventually maintain consistent with the leader is it does not die.

The Byzantine Generals Problem:

In fault-tolerant computer systems, and in particular distributed computing systems, Byzantine fault tolerance is the characteristic of a system that tolerates the class of failures known as the Byzantine Generals' Problem(described by Leslie Lamport, Robert Shostak and Marshall Pease in their 1982 paper, "The Byzantine Generals Problem") , which is a generalized version of the Two Generals' Problem.

The Byzantine Generals Problem is an abstraction of the problem of reaching an agreement in a system where components can fail in an arbitrary manner. In such a case, the component can behave arbitrarily and can send different messages to different components. The abstraction of the problem deals with the idea of generals of the Byzantine Army communicating with each other. The generals must reach a consensus among themselves whether to attack or retreat based on the messages exchanged. The problem is complicated by the fact that some of the generals can be traitors who may send conflicting messages to the other generals. The solution to the problem must allow all the loyal generals to agree upon a common plan of action. Also, if the commanding general is loyal then all the loyal generals must obey the order he sends.

2.2 Related research to solve the problem

We have not found the related papers about the Byzantine fault tolerance solution in the Raft. We decide to study the solutions for the Byzantine problem and adopt them into the implementation of the Raft.

Traditional Byzantine-fault-tolerance protocol, or oral message, introduced in *an Optimal Probabilistic Protocol for Synchronous Byzantine Agreement* by Prsech and Silvio in 1997, is trying to solve this problem by sending knowledge of others received message to each other. The principle of this protocol is during every round of sending, each node would send the previous information of others, which received by this node, to everyone else in the cluster. According to the configuration of the choice, the total number of round needed is deterministic. To achieve a total number of F faulty nodes, the whole cluster would need at least total number of 3F+1 nodes. And the total number of information exchange round needed is at least F+1.

Signed message algorithm: In the above solution, the time complexity is O(n^m) for m faulty nodes, which is very expensive. Another solution is with signed messages. In this algorithm, each general can send only unforgeable signed messages. There are two assumptions: (1) a loyal general’s signature cannot be forged; (2) anyone can verify authenticity of general’s signature. Therefore, if commander(leader) is not faulty, then non-faulty nodes can verify its identification and get that correct message. If messages sent from faulty nodes are forged, non-faulty nodes can verify that they are not sent by commander(leader). Non-faulty nodes can still get the same messages to follow. In another case, if commander is faulty, it might send different messages to all nodes. Then after verification, non-faulty nodes will find that messages received by itself and by others are not the same, then it will do nothing. As long as non-faulty nodes do nothing(the same thing), consistent is preserved. This will prevent a traitor general from sending a value other than what he receives.

Practical Byzantine fault tolerance: In 1999, Miguel Castro and Barbara Liskov introduced the "Practical Byzantine Fault Tolerance" (PBFT) algorithm, which provides high-performance Byzantine state machine replication, processing thousands of requests per second with sub-millisecond increases in latency. PBFT triggered a renaissance in Byzantine fault tolerant replication research, with protocols like Q/U, HQ, Zyzzyva, and ABsTRACTs working to lower costs and improve performance and protocols like Aardvark and RBFT working to improve robustness.

Byzantine fault tolerance in practice: One example of BFT in use is Bitcoin, a peer-to-peer digital currency system. The Bitcoin network works in parallel to generate a chain of Hashcash style proof-of-work. The proof-of-work chain is the key to overcome Byzantine failures and to reach a coherent global view of the system state. Some aircraft systems, such as the Boeing 777 Aircraft Information Management System (via its ARINC 659 SAFEbus® network), the Boeing 777 flight control system, and the Boeing 787 flight control systems, use Byzantine fault tolerance. Because these are real-time systems, their Byzantine fault tolerance solutions must have very low latency.

2.3 Advantage/ disadvantage of those research

The Traditional Byzantine Fault Tolerance Protocol requires a lot of communication when we want more fault tolerance. According to the paper, this method would send O(nF+1) of message to make sure the whole system consensus. The consequence of this large amount of communication is the bad scalability. When the number of nodes in the cluster increase, and even we want to achieve higher fault tolerance of faulty node, the message would make the system have a bad performance.

Signed message algorithm can handle m faulty nodes with any number of nodes. If n = m + 1, no consistency problem. If n >= m + 2, the non-faulty nodes will do the same thing. Also, since only unforgeable signed messages are sent the number of messages exchanged is smaller than Traditional Byzantine Fault Tolerance Protocol.

Besides, this message communication is built on the assumption that all nodes have direct link with each other in the cluster, which might be a problem when implement this protocol into some real situation. And these huge amount of communication would be unnecessary and inefficient when there is no faulty node among the cluster.

2.4 Our solution to solve this problem

We found there are two different solutions for Byzantine problem, oral message and signed message. We choose solution of signed message and integrate it into Raft to make Raft can tolerate Byzantine failure. In order to integrate signed message algorithm into Raft, we make following modification:

1. Applying asymmetric cryptography on each communication between each node: we use asymmetric cryptography to implement unforgeable message signature which is required in signed message algorithm.
2. Adding additional communication between followers: in Raft, followers follow commands from a leader, such as append a log entry or commit a log entry. A follower only communicate with other followers in the leader election. However, to reach Byzantine fault tolerance through using signed message algorithm, a followers need to tell other followers what message he received from a leader. Hence, we add additional

RPC for the use of sending such messages.

2.5 Why our solution is better

Raft assumes that nodes fail only by stopping, which rarely holds in practice unfortunately. Malicious attacks and software errors can cause faulty nodes to exhibit Byzantine behavior, consequently making the replica inconsistent, in the end, the new leader might not retrieve the latest user input because it can not get the quorum vote among the followers. We aim to enhance the original Raft algorithm such that it becomes tolerant to Byzantine server behaviors.

3. Hypothesis/Goal

Our hypothesis is that we think a Byzantine leader sends different user requests or send commit requests among followers, the log file will become inconsistency among followers.

4. Methodology

4.1 How to generate/ collect input data

Our input data are some client requests designed by ourselves to change states of each replica.

4.2 How to solve the problem

We will implement a Byzantine fault tolerant Raft system through signed message algorithm to solve Byzantine leader problem in java. This system is a distributed system and user can send request to this system to change its state. We also provide two commands, one for enable/disable simulation of Byzantine failure and the other one for enable/disable Byzantine fault tolerance. This system records every state change and all communications happened in the system. We can verify if our solution successfully handle Byzantine leader with these records.

To solve Byzantine problem, first, we introduce asymmetric cryptography into Raft. Each node maintains three types of key: its private key, other followers' key and leader's key. Before sending a message to any other node, a node needs to encrypt the message with its private key. On the other hand, once a node receives an encrypted message, it can decrypts the message with a public key corresponding to the sender. Thus, we can ensure that a message received by a node is not modified by any other node. Second, we modify Raft protocol to make a follower receiving a message sent by leader encrypts and forwards the message to other followers. A follower receiving a message from other followers uses followers' public keys to decrypt it first and than using leader's public key to decrypt it again. By this way, a follower knows what messages are leader sent to other followers and if these messages are identical. If majority of messages are identical, then followers execute these messages. Otherwise, followers simply ignore these messages. Using asymmetric cryptography is also prevent a fake leader, a follower acts as a leader and sends messages to other nodes. When receiving a message which is claimed from a leader, a follower tries to decrypt it by using leader’s public key. If fail to decrypt the message, the follower knows it comes from a fake leader and discards it.

4.3 How to generate output

Our output data stored on each node contain following things:

1. state change
2. record of message
    a. received or sent
    b. sender or receiver
    c. timestamp.

4.4 How to test against hypothesis

We will send a sequence of request to the system with two different setup:

1. enable Byzantine failure and disable Byzantine fault tolerance
2. enable Byzantine failure and enable Byzantine fault tolerance

and check their output.

5. Implementation

5.1 Code implementation

5.1.1 Code source

Codes are all original codes by all team members. We neither use open source code, nor use open source code as reference.

5.1.2 Language

We use Java to implement codes. JDK 1.8.

5.1.3 Class Diagram

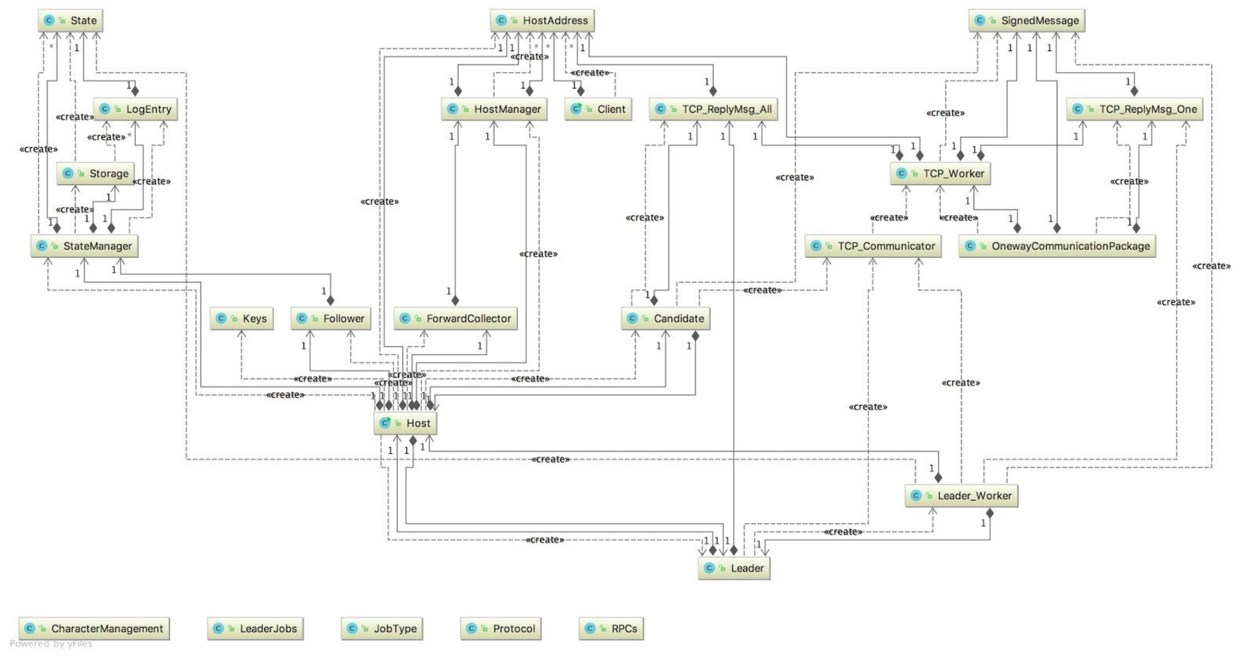


Figure 1 All the class implemented in the enhanced RAFT. Up right corner classes controls the state machine. Up right corner classes controls the signed communication. Center classes controls the RAFT character switching.

5.1.4 Key Components

Client Interface

The client interface acts both the cluster coordinator and the RAFT user. It is like the command line program.

The client would build up the whole cluster and make all host connected according to the terminal input. It is also responsible for sending the instruction to ask the cluster activating or deactivating the enhanced Byzantine fault tolerance functionality.

The client receives user input and send the user command to the leader in the cluster. According to the optional flag, it would tell the leader whether this command needs it to do a Byzantine behavior.

____

(Sample Code)

```
System.out.print("Client => ");
userInput = in.nextLine();
userInput.trim();
String cmdCode = decodeCommand(userInput);
if (Objects.equals(cmdCode, "add")) {
  serverInfos = addMultipleParser(userInput);
  if (serverInfos.isEmpty()) {
     System.out.println("no host has added");
     continue;
  }
  HostAddress s = serverInfos.get(0);
  try{
     Socket socket = new Socket(s.getHostIp(), s.getHostPort());
     ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
     outStream.flush();
     outStream.writeInt(Protocol.ADDHOSTADDRESS);
     outStream.flush();
     outStream.writeObject(serverInfos);
```

```
        outStream.flush();
        ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
        if(inStream.readInt() != Protocol.ACKOWLEDGEMENT){
          System.out.print("ACK NOT RECEIVED\n");
          // maybe need to try again
        }
        socket.close();
      }catch (IOException e){
        System.out.println("Please check the server is active or key in the correct address and port.");
        System.out.print("Failed on server ");
        System.out.print(s.getHostIp());
        System.out.print(", port number ");
        System.out.print(s.getHostPort());
        System.out.println(".");
      }
    }
```

____

Leader, Follower, Candidate

These three class would contain all the job it would do. The leader would send append entry request to all the followers (heartbeat is a special append entry request with the append value empty). The follower would respond to the leader's request and send back acknowledgement or disacknowledgement according to whether it has commit the log entry into the disk. The goal of the candidate is trying to become a new leader of the cluster. It would keep sending vote requests until it receives enough vote to become new leader or someone has become new leader and sends heartbeat message to it.

____

(Sample Code)

```
_votes = 1;
HashMap<String, Thread> threads = new HashMap<>();
```

```
int queueSize = _queue.size();
boolean appendFlag = false;
for(String hostname : _hostnames){
  if(!hostname.equals(_host.getHostManager().getMyHostName())){
    if(!_isFindNextIndex.contains(hostname)){
      threads.put(hostname, new Thread(new Leader_Worker(this, LeaderJobs.FINDINDEX, hostname, _host)));
    }else if(_nextIndex.get(hostname) <= _host.getCommitIndex()){
      threads.put(hostname, new Thread(new Leader_Worker(this, LeaderJobs.KEEPUPLOG, hostname, _host)));
    }else if(queueSize > 0){
      appendFlag = true;
      threads.put(hostname, new Thread(new Leader_Worker(this, LeaderJobs.APPENDLOG, hostname, _host)));
    }else{
      threads.put(hostname, new Thread(new Leader_Worker(this, LeaderJobs.HEARTBEAT, hostname, _host)));
    }
    threads.get(hostname).setDaemon(true);
    threads.get(hostname).start();
  }
}
```
____

State Machine

The state machine is responsible for managing all the log entry, all the functionality of its host. The cluster coordinator (client interface) would send request to the host, and host would change the flag in the state machine, Byzantine enable flag, for example. The host behavior would follow these flags. The file storage management unit is also handled by the state machine. The followers and leader would add, delete and commit a log entry through the state machine management.

____

(Sample Code)

```
public synchronized boolean commitEntry(int at) {
    LogEntry logToCommit = stateLog.get(at);
    if (logToCommit == null) {
      System.out.println("no found");
      return false;
    }
    if (commitFailEnable) {
      return false;
    }else if (fileStoreHandler.storeNewValue(logToCommit)){
      LogEntry temp = stateLog.get(at);

states.get(stateLog.get(at).getState().getStateName()).changeState(stateLog.get(at).getState().getStateValue());
      states.get(temp.getState().getStateName()).changeState(temp.getState().getStateValue());
      stateLog.get(at).commitEntry();
      return true;
    }else {
      return false;
    }
  }
```

____

Data Storage

Each host would manage two files, log file and vote file, both are txt file.

Log file would store all the log entries, which have been committed by the host already. The newest committed entry would stored in the last line of the file. During the startup of the host machine, it would read the existing log file in the disk, if exists, and loads all the entry in the file into the memory. This could prevent data loss when one host is crashed. During the running time, the host might need to delete the already committed entry, which is caused by the leader synchronization. Also, it could append new entry into the file.

Vote file would store the vote information during the leader election period. The candidate would send multiple vote requests until it gets enough vote to decide whether it would become a leader or all the unvoted host reach the internal timeout. When a host crashes just after it sends the vote and recover before all candidate become new leader, this vote storage would prevent a host sending multiple vote to the candidates, which could prevent more than 1 candidate gets majority vote during a single leader election period.

____

(Sample Code)

```
if (new File("./" + hostName).mkdir()) {
}else {
}

logFilePath = "./" + hostName + "/logFile.txt";
voteFilePath = "./" + hostName + "/voteFile.txt";
boolean newFile = false;
try{
  logFile = new File(logFilePath);
  if (!logFile.exists()) {
     logFile.createNewFile();
```

```
        System.out.println("create file");
        newFile = true;
    }
}catch (IOException exception){
    System.out.println("log file open failed");
    exception.printStackTrace();
}
try {
    logFileWrite = new FileWriter(logFile, true);
}catch (IOException exception) {
    System.out.println("open file writer");
    exception.printStackTrace();
    System.exit(0);
}
if (newFile) {
    try {
        logFileWrite.flush();
        logFileWrite.write("Index\tTerm\tVariableName\tValue");
        logFileWrite.flush();
        logFileWrite.write(System.lineSeparator());
        logFileWrite.flush();
    }catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

____

Signed Message

Each host has a set of Public Key and Private Key when the host is started. When hosts are adding each other to form a cluster, each host sends its Public Key to all other hosts. Each host maintain other hosts' ip address, port number, and public key.

____

(Sample Code)

```
public Keys() {
  KeyPairGenerator keyPairGen = null;
  try {
     keyPairGen = KeyPairGenerator.getInstance("RSA");
  } catch (NoSuchAlgorithmException e) {
     e.printStackTrace();
  }
  // init key size to be 1024
  keyPairGen.initialize(512);

  // generate key pair
  KeyPair keyPair = keyPairGen.generateKeyPair();
  // get private key
  this.privateKey = (RSAPrivateKey) keyPair.getPrivate();
  // get public key
  this.publicKey = (RSAPublicKey) keyPair.getPublic();
}
```

____

When it becomes to leader, it uses its Private Key to encrypt the message and send out. When someone receive a signed message, it first tell who sends this message. Then it resorts to HostManager, where other hosts' information is stored, to find out sender's Public Key. And then use the Public Key to decrypt the signed message to get original message.

____

(Sample Code)

```
public static byte[] encrypt(Key k, String data) {
  byte[] data_bytes = new byte[0];
  try {
        data_bytes = data.getBytes("UTF-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    if (k != null) {
        Cipher cipher = null;
        try {
            cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.ENCRYPT_MODE, k);
            byte[] resultBytes = cipher.doFinal(data_bytes);
            return resultBytes;
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (NoSuchPaddingException e) {
            e.printStackTrace();
        } catch (BadPaddingException e) {
            e.printStackTrace();
        } catch (IllegalBlockSizeException e) {
            e.printStackTrace();
        } catch (InvalidKeyException e) {
            e.printStackTrace();
        }
    }
    return null;
}
```

____

TCP Communication & Timeout

All communications between hosts, including broadcast and one-to-one communication are TCP communications. Basic communication unit is set to be one round, which contains one send and one receive. If one initiates one communication with one other host, it waits for a reply. If one receive a message from one other host, it must give a reply to that host to help it to finish this one-round communication.

However, none host will wait forever, since timeout is enforced. Whoever initiates one communication, it starts a new thread to do the real socket communication, the main thread is waiting for a certain amount time(eg 500ms), then the program goes on to next step to check if there is reply. Two possible results may happen. One is that there is reply. Then host uses the reply to go on. The other one is that there is no reply or false reply. Then host will regard that receiver fails and go on, no matter due to network congestion or receiver is down.

____

(Sample Code)

```
public boolean initSendToOne(HostAddress targetHost, TCP_ReplyMsg_One tcp_ReplyMsg_One, SignedMessage
msg) {
  TCP_Worker worker = new TCP_Worker(targetHost, tcp_ReplyMsg_One, msg, targetHost.getPublicKey(),
JobType.sentToOne);
  worker.start();
  if (DEBUG) System.out.println("From communicator: worker started with " + targetHost.getHostName() + ", " +
targetHost.getHostIp());

  try {
     Thread.sleep(300);
  } catch (InterruptedException e) {
     e.printStackTrace();
  }
  return tcp_ReplyMsg_One.getMessage() != null;
}
```

____

5.2 Design document and flowchart

Leader Election

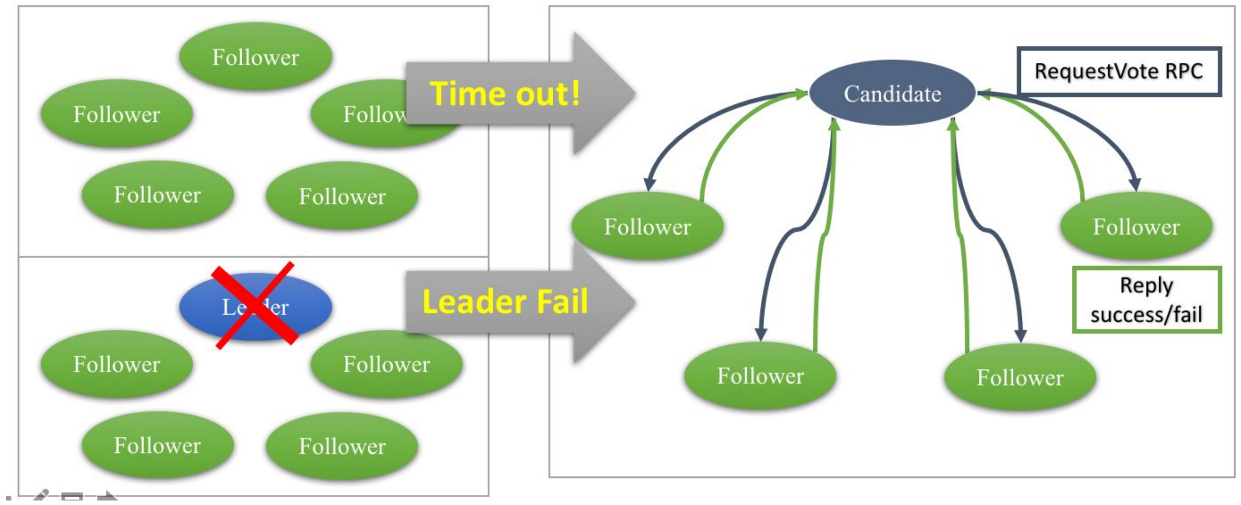


Figure 2 Leader election process in Original Raft.

As shown in Figure 2, when one of the follower times out (Initial set up or leader fail), it becomes to be a Candidate, and sends RequestVote RPC to other followers. Followers will send success back to the sender if its current term is lower than the sender's term. If the Candidate receives the majority votes, it will become the leader in the system.

Normal RAFT

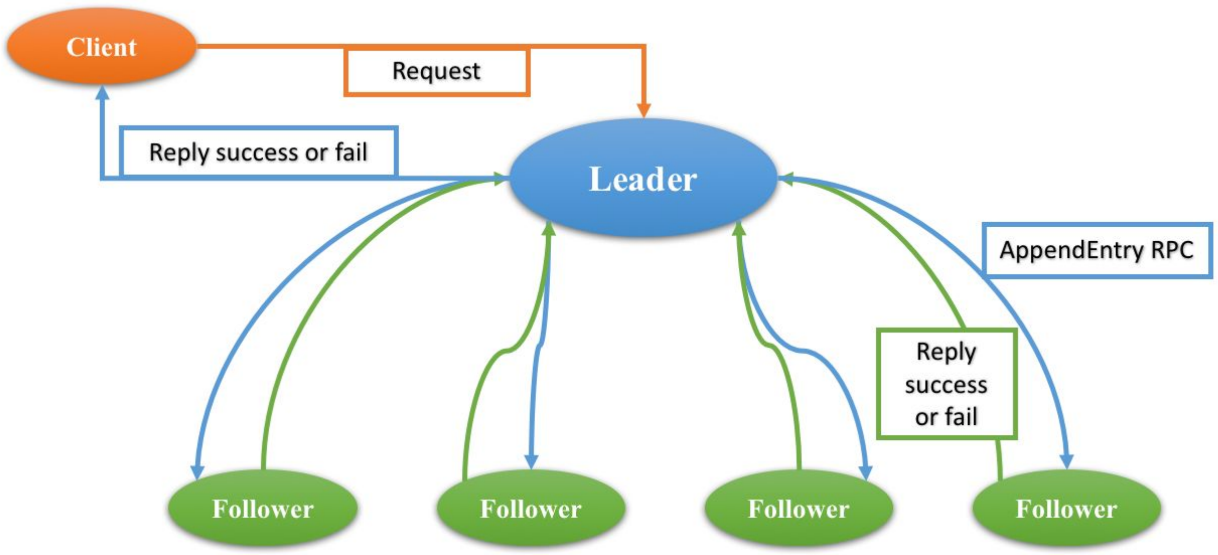


Figure 3 In normal usage condition for RAFT.

As shown in Figure 3, the Leader will accept the request from the client, then, send appendEntry RPC to all of the follower, followers will send success/fail back to the leader if the entry satisfy the RAFT append rules, leader will reply success to client if it gathers the majority agreement.

Byzantine RAFT

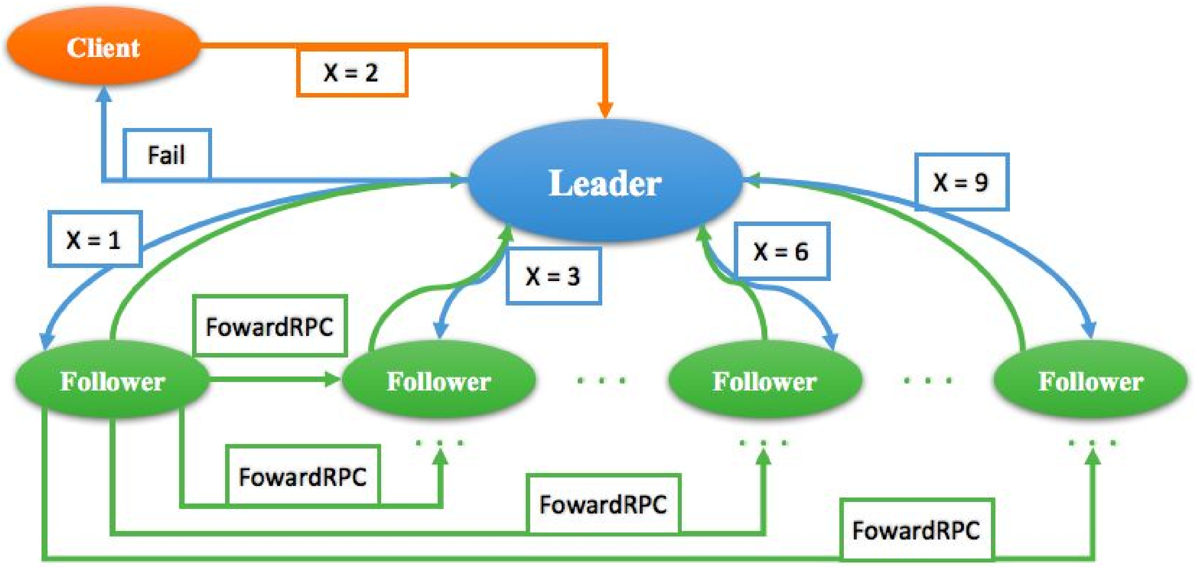


Figure 4, Byzantine case for RAFT with Enhanced RAFT.

As shown in Figure 4, in Byzantine case for RAFT, the Leader will change the request value that Client sends, and may also send different value to each follower. We simulate these different values that leader send by letting Leader to pick a random value. Now Follower doesn't trust what leader says. So Follower will verify if Leader is a Byzantine Leader. Follower will forward the message that it received to all other followers. Then each follower will collect all unique messages it receives from other followers. If there is a message that whose count number is over majority, then Follower will adopt this value and commit. Since we use signed message, Follow doesn't have the ability to fake a message which is signed by Leader. Also, if one Follower tries to change what it received from Leader then forward, the receiver will distinguish that it's not real message send by user and will discard it. Since all Followers will do same thing, consistency will be maintained.

Flowchart: (Skeleton)

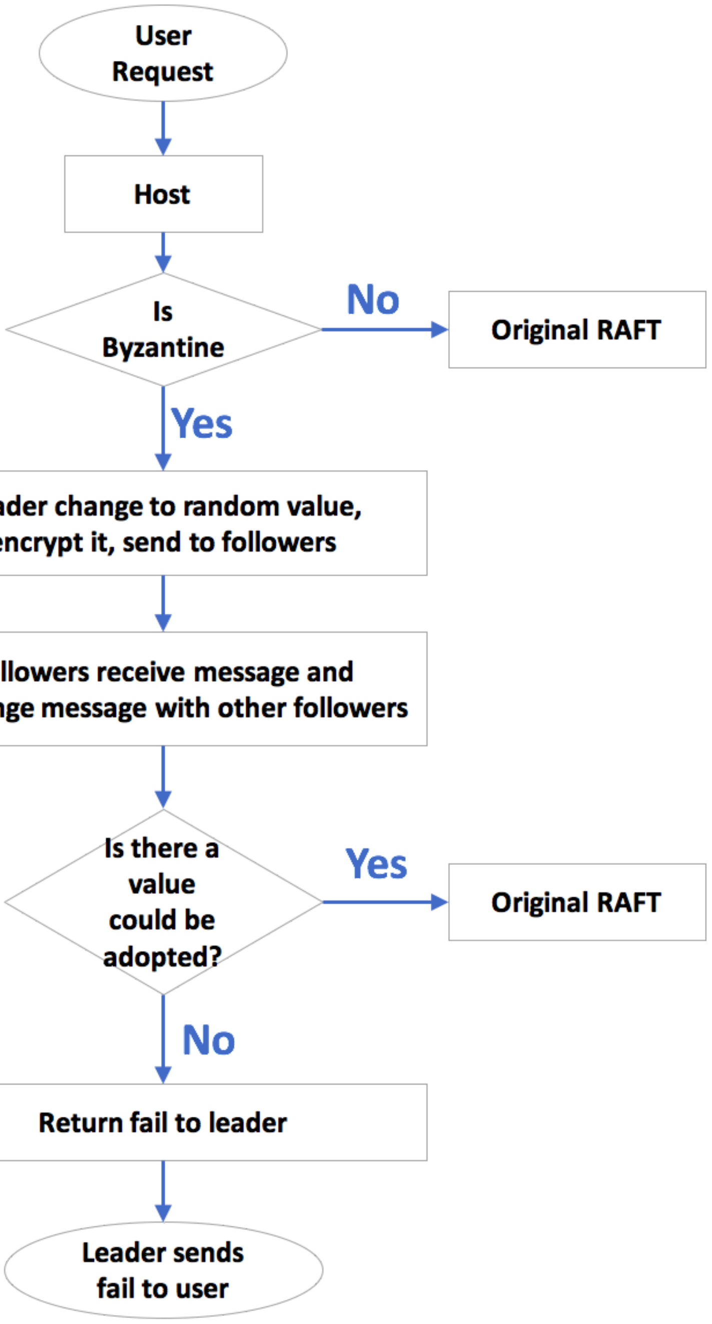

User Request
Host
Is Byzantine
No
Original RAFT
Yes
Leader change to random value, encrypt it, send to followers
Followers receive message and exchange message with other followers
Is there a value could be adopted?
Yes
Original RAFT
No
Return fail to leader
Leader sends fail to user

Figure 5 Flowchart for Enhanced RAFT.

As shown in Figure 5, this is a skeleton flowchart of how this program works to test that we solve Byzantine Leader. Many detailed flows are hidden from flowchart to show a clear whole view. User request will be first received by Host. Host will judge that if this request is under Byzantine situation or not. If it’s not under Byzantine situation, show original RAFT functions. If this request is under Byzantine situation, we will open functions which support resolving Byzantine. This check is only for testing purpose. We can show the consistency it’s original RAFT and show inconsistency that RAFT can’t solve.
Then in Byzantine situation, Leader will send other value than what Client want it to send to Followers. Followers receive the message and exchange with other Followers. They they will conclude that which value should be adopted or this request should be discarded. If there is, ll followers will commit like original RAFT. Otherwise, they will return fail to leader. Leader will notify Client that this request fails.

6. Data analysis and discussion

6.1 Output generation

Our enhanced raft system stores log on each host as output date file. User can apply instructions to enhanced raft system to change state of each host. These changes will be saved in the log of each host and stored as output data file. Enhanced raft system provides following instructions:

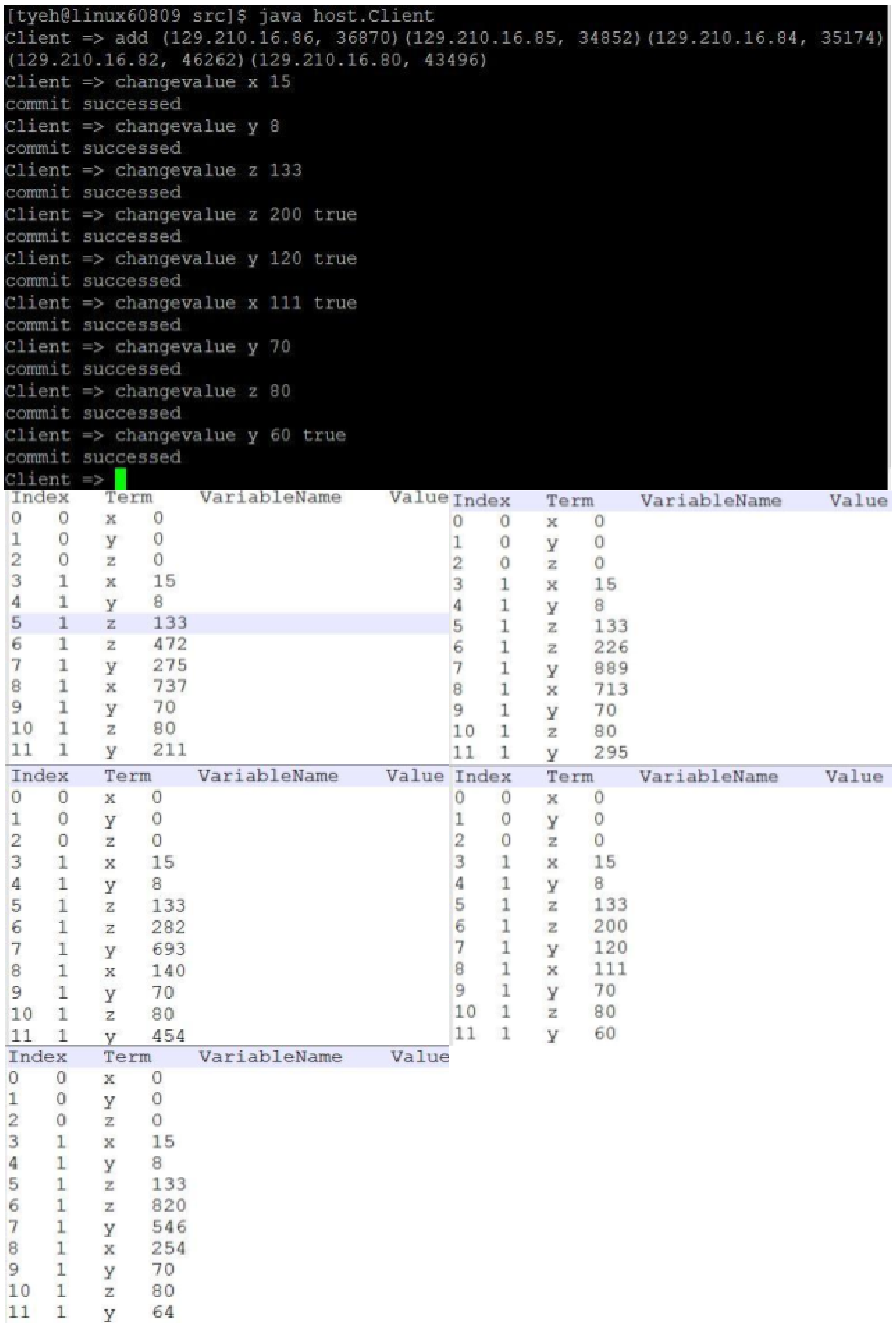
1. add (<host ip: String>, <host port: String>) …: This instruction would connect all the host into one cluster and start the remaining functionality of the RAFT.
2. byzantineenable / byzantinedisable: These two instruction would activate/deactivate the enhanced Byzantine fault tolerance functionality in all host machines.
3. changevalue <state name: String> <state value: Int> <?byzantine command: Bool>: This instruction would send the new value with the new state name to the leader. The first parameter is key word "changevalue"; the second parameter is state name, which is a

string without empty space; the third parameter is new state value, a integer; the last parameter is an optional Bool value, true stands for the leader would make Byzantine move on this command.

4. help: Cheat sheet of all the instructions.

6.2 Output analysis

We clustered five different hosts as our enhanced raft system. Enhanced raft system is initiated with three states, x, y, and z, which values are zero. We run enhanced raft system with two different setups. The first setup is disable signed message algorithm. When signed message algorithm is disabled, enhanced raft system works same as original raft. Figure 6 shows constructions applied and log recorded. We applied 4 instructions to trigger byzantine leader. For output files, we can see each file has different value on index 6, 7, 8 and 10. Indices of inconsistent log entry are match with instructions. This proof that original raft encounters inconsistency when leader is byzantine fail.



```
[tyeh@linux60809 src]$ java host.Client
Client => add (129.210.16.86, 36870)(129.210.16.85, 34852)(129.210.16.84, 35174)
(129.210.16.82, 46262)(129.210.16.80, 43496)
Client => changevalue x 15
commit successed
Client => changevalue y 8
commit successed
Client => changevalue z 133
commit successed
Client => changevalue z 200 true
commit successed
Client => changevalue y 120 true
commit successed
Client => changevalue x 111 true
commit successed
Client => changevalue y 70
commit successed
Client => changevalue z 80
commit successed
Client => changevalue y 60 true
commit successed
Client =>
```

```
Index   Term    VariableName    Value
0   0   x   0
1   0   y   0
2   0   z   0
3   1   x   15
4   1   y   8
5   1   z   133
6   1   z   472
7   1   y   275
8   1   x   737
9   1   y   70
10  1   z   80
11  1   y   211
```

```
Index   Term    VariableName    Value
0   0   x   0
1   0   y   0
2   0   z   0
3   1   x   15
4   1   y   8
5   1   z   133
6   1   z   226
7   1   y   889
8   1   x   713
9   1   y   70
10  1   z   80
11  1   y   295
```

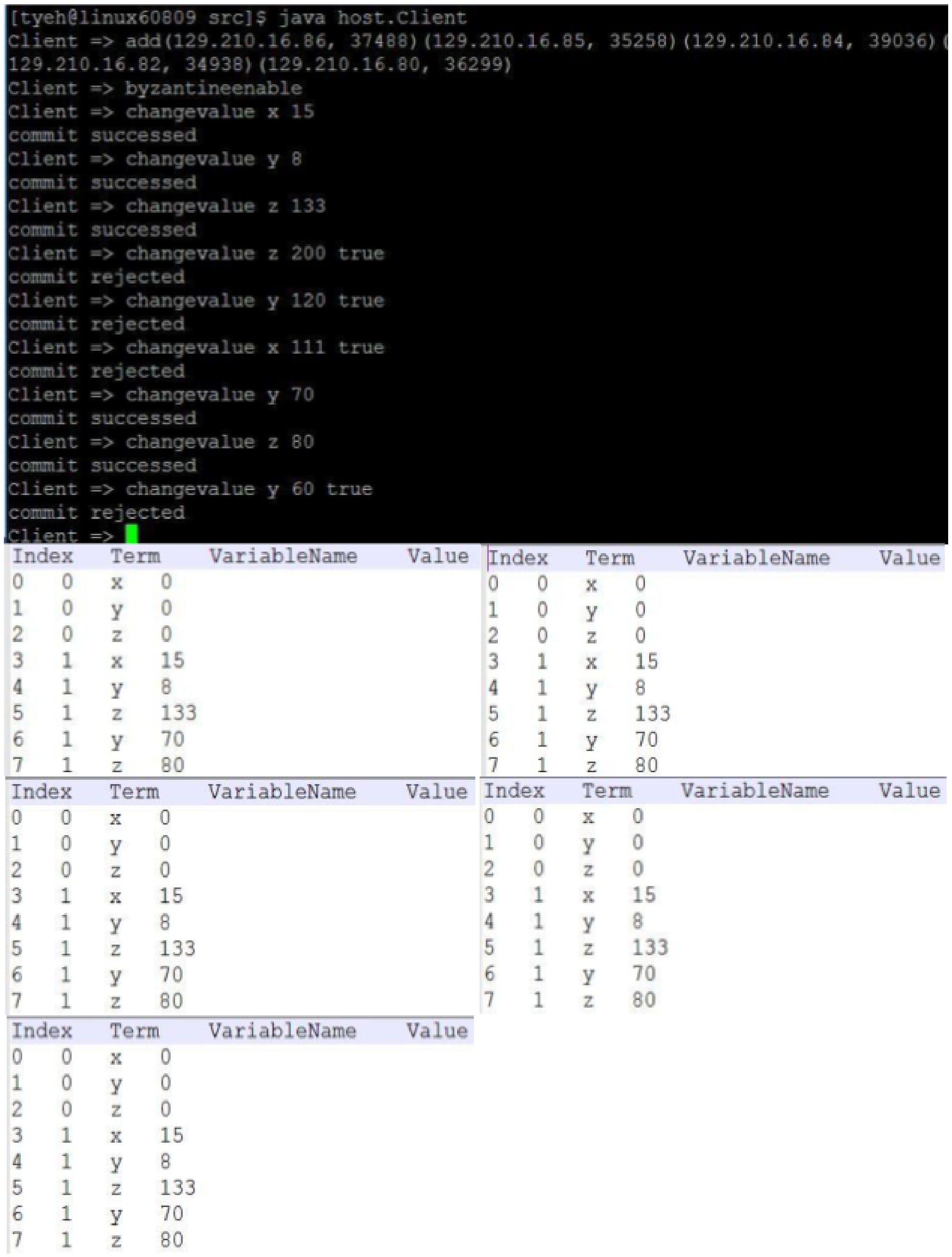
```
Index   Term    VariableName    Value
0   0   x   0
1   0   y   0
2   0   z   0
3   1   x   15
4   1   y   8
5   1   z   133
6   1   z   282
7   1   y   693
8   1   x   140
9   1   y   70
10  1   z   80
11  1   y   454
```

```
Index   Term    VariableName    Value
0   0   x   0
1   0   y   0
2   0   z   0
3   1   x   15
4   1   y   8
5   1   z   133
6   1   z   200
7   1   y   120
8   1   x   111
9   1   y   70
10  1   z   80
11  1   y   60
```

```
Index   Term    VariableName    Value
0   0   x   0
1   0   y   0
2   0   z   0
3   1   x   15
4   1   y   8
5   1   z   133
6   1   z   820
7   1   y   546
8   1   x   254
9   1   y   70
10  1   z   80
11  1   y   64
```

Figure 6 The implementation for Enhanced RAFT at normal case.

As shown in Figure 7, the second setup is enable signed message algorithm. We applied same instructions as first setup. Next figure is the result of second setup. From the figure, we can see constructions triggering byzantine leader were rejected and all output file stay consistent. This proofs signed message algorithm applied to Raft can solve byzantine leader problem.



```
[tyeh@linux60809 src]$ java host.Client
Client => add(129.210.16.86, 37488)(129.210.16.85, 35258)(129.210.16.84, 39036)(
129.210.16.82, 34938)(129.210.16.80, 36299)
Client => byzantineenable
Client => changevalue x 15
commit successed
Client => changevalue y 8
commit successed
Client => changevalue z 133
commit successed
Client => changevalue z 200 true
commit rejected
Client => changevalue y 120 true
commit rejected
Client => changevalue x 111 true
commit rejected
Client => changevalue y 70
commit successed
Client => changevalue z 80
commit successed
Client => changevalue y 60 true
commit rejected
Client =>
```

Index	Term	VariableName	Value
0	0	x	0
1	0	y	0
2	0	z	0
3	1	x	15
4	1	y	8
5	1	z	133
6	1	y	70
7	1	z	80

Index	Term	VariableName	Value
0	0	x	0
1	0	y	0
2	0	z	0
3	1	x	15
4	1	y	8
5	1	z	133
6	1	y	70
7	1	z	80

Index	Term	VariableName	Value
0	0	x	0
1	0	y	0
2	0	z	0
3	1	x	15
4	1	y	8
5	1	z	133
6	1	y	70
7	1	z	80

Index	Term	VariableName	Value
0	0	x	0
1	0	y	0
2	0	z	0
3	1	x	15
4	1	y	8
5	1	z	133
6	1	y	70
7	1	z	80

Index	Term	VariableName	Value
0	0	x	0
1	0	y	0
2	0	z	0
3	1	x	15
4	1	y	8
5	1	z	133
6	1	y	70
7	1	z	80

Figure 7 The implementation for Enhanced RAFT at byzantine case.

7. Conclusion and recommendations

7.1 Summary and conclusions

By using signed message transmission and forwarding the leader message, we are able to make the RAFT immune to a Byzantine leader, which would arbitrary change the user command and send to the followers. The original RAFT would become inconsistent when a Byzantine leader shows up, because it is relying the assumption that once the index and the term are the same, the value in that log entry must be the same. Our enhanced RAFT would solve this inconsistency by forwarding the leader's command to all other followers once.

The overhead of our method is on the communication part. We need to forward a total number of $O(N^2)$ messages across the cluster when the leader sends one command to the followers, where N represents the number of hosts in the cluster.

7.2 Recommendations for future studies

Our project is based on some assumption, in order to testify our method could solve the Byzantine leader fault. In the real environment, these assumption might not be met, which would make our project unsuitable for the real world system.

We have assumed only the leader would become the Byzantine node, and the followers would not. In the real world, a Byzantine followers might become silence, send fault commit signal to the leader, vote to an unsuitable candidate or vote to multiple candidates during the leader election period. In the further improvement, we might need to come up with some solutions to make the RAFT become fully tolerated to arbitrary Byzantine node.

Until now, our Byzantine enhanced RAFT could assure the whole cluster would become consistent when a Byzantine leader shows up. However, we can not assure the committed command is the same as the user typed command. This might become a problem because it would act like the cluster has committed an unauthorized command, and the source of the command is unknown, because no client but the Byzantine leader is responsible to it. In the future implement we might need to fix this inconsistency between the client and the cluster.

Furthermore, we only tested one kind of the Byzantine leader behavior, sending arbitrary message to the followers. One possible Byzantine behavior a leader might do is keeping increasing the current term. This would not affect the consistency of the cluster because the original RAFT would solve this problem. However, it would strongly make the cluster do unnecessary work load. We need a mechanism to detect this kind of behavior and may kick the current leader into a follower. Another possible Byzantine behavior is that the leader might not send user's command to the followers while it would still tell the user that the command has been committed.

8. Bibliography

[1] Ongaro, Diego, and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm." USENIX Annual Technical Conference. 2014.

[2] Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." ACM Transactions on Programming Languages and Systems (TOPLAS) 4.3 (1982): 382-401.

[3] Driscoll, Kevin, et al. "The real byzantine generals." Digital Avionics Systems Conference, 2004. DASC 04. The 23rd. Vol. 2. IEEE, 2004.

[4] Feldman, Paul, and Silvio Micali. "An optimal probabilistic algorithm for synchronous byzantine agreement." Automata, languages and programming (1989): 341-378.

[5] Lamport, Leslie. "Paxos made simple." ACM Sigact News 32.4 (2001): 18-25.

[6] Moraru, Iulian, David G. Andersen, and Michael Kaminsky. "Paxos quorum leases: Fast reads without sacrificing writes." Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.

[7] Marandi, Parisa Jalili, et al. "The performance of Paxos in the cloud." Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on. IEEE, 2014.