華中科技大學

A Simple Crash-tolerant Consensus Implementation Using Raft

Team Member 1:	Sixu Hu（胡思勖）
Email address:	husixu1@hotmail.com
Team Member 2:	Yifu Deng（邓一夫）
Email address:	deng3652@gmail.com
Team Member 3:	Junhua Zhang(张俊骅)
Email address:	497387417@qq.com

2018.01.01

Contents

Chapter 1

Raft Algorithm Introduction

1.1 Consensus protocol

Consistency means that multiple servers agree in state, but in a distributed system, because of various accidents, some servers may crash or become unreliable, and it can not reach a consistent state with other servers. This requires a consensus protocol, which is designed to ensure fault tolerance, that is, even if there are one or two servers in the system that do not impact its processing.

1.2 Raft algorithm

In a distributed database system, if each node has the same initial state and each node performs the same sequence of operations, then they eventually get a consistent state. To ensure that each node executes the same sequence of commands, it is necessary to execute a ”consistency algorithm” on each instruction to ensure that the instructions seen by each node are consistent. And Raft is such an algorithm, we can use it to solve the problems.

1.3 The protocol of Raft

- Each replica of the agreement is in one of three states: Leader, Follower or candidate.
- Leader is the processor of all requests, the replication of leader receives the client's update request, do local process and then synchronized to other copies.
- Follower is passive updaters of the request, it receives update requests from leader, and then write to local logs. A follower does not take the initiative to initiate the request.
- Candidate is the intermediate state from follower to leader conversion. when the follower replica does not receive a heartbeat from the leader replica for a period of time, then it shows that the leader may

have failed. At this time, the election process is started and the replica becomes a candidate until the election is over.

1.4 The details of Raft

1.4.1 Terms

As we all know, in a distributed environment, time synchronization is a big problem, but in order to identify outdated information, time information is essential. To solve this problem, in raft algorithm, the time is divided into terms, a term can be considered a logical time.

- There is at most one leader per term.
- In some terms, due to the failure of the election, there is no leader.
- Every servers maintains current term locally.

1.4.2 Heartbeats and Timeouts

- All servers start as a follower, and start the election timer.
- Follower expects to receive RPC from leader or candidates.
- Leader must broadcast heartbeat to reset follower's election timer.
- If the follower' s election timer times out, it assumes that the leader has crashed and then initiates a election.

1.4.3 Leader election

Increment by term, follower changes to candidate, set a vote for itself, initiate a RPC of requesting vote in parallel, and try again until one of the following conditions is met:

- Get over half votes of servers, then convert to leader, broadcasts heartbeat simultaneously.
- Reveive legal RPC of append entries, then converted to follower.
- Election timeout, no successful server election, increment by term and election again.

1.4.4 Log replication

The normal process:

- Client sends command to the leader.

- Leader appends command to the local log.
- Leader broadcasts the append entries RPC to followers.
- Once the log entry committed successfully:

  (1) The leader applies the corresponding command to the local state machine and returns the result to the client.

  (2) The leader notifies committed log entries to follower through subsequent append entries RPC.

  (3) If the follower receives the commit log entry, applies it to the local state machine.

1.4.5 Safty

In order to ensure the correctness of the whole process, the Raft algorithm guarantees that the following attribute are true:

- Election Safety

  Within any given term, select up to one leader.

- Leader Append-Only

  The leader never rewrites or deletes the local log, just appends the local log.

- Log Matching

  If the log entries on both nodes have the same index and term, the logs in the two nodes [0, Index] are exactly the same.

- Leader Completeness

  If a log entry is committed in a term, then any subsequent term of leader owns the log entry.

- State Machine Safety

  Once a server applies a log entry to a local state machine, all subsequent server apply the same log entry for that offset.

1.4.6 Other details

- Candidate may receive append entries RPC from other leaders while waiting for the voting result. If the leader' s term is not less than the local current term, the legitimacy of the leader identity is recognized and proactively downgraded to follower; otherwise, the candidate identity is maintained and wait for a voting result.

- Candidate has neither successful election nor received RPC from other leaders. It usually happens when multiple nodes initiate the election at the same time, and eventually each candidate will be timeout. In order to reduce the conflict, we shall use a strategy named random concession, it means each candidate restart the election timer as a random value, and then it greatly reduced the probability of conflict.

Chapter 2

Difference between Raft & Paxos

2.1 Brief discription of two algorithms

- Raft is a consensus algorithm for managing a replicated log designed by Diego Ongaro and John Ousterhout, Stanford University. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems.
- Paxos and Raft are called consensus protocols. As the name implies, they address the issue of multi-node consensus. It should be noted that the consensus mentioned here does not mean that all nodes are in exactly the same state at any moment, but to ensure that: even in the event of network partition or machine node anomalies, the entire cluster can still provide a consistent service, that is, each operation can see that all previous successful operations are completed in sequence.

2.2 Different difficulty

The advantages of the Raft protocol over Paxos are that Raft is easier to understand and implement. It strengthens the leader's position, and the entire protocol can be clearly divided into two parts, and uses the continuity of the log to do some simplification:

- When the leader is running normally, it synchronizes the log of followers.
- When the leader is crashed, choose a new leader.

2.3 Different process

Multi-paxos and Raft both use a number to identify the leader, multi-paxos called proposer-id, Raft called term, the meaning is the same. In Multi-paxos, the leader which has the biggest proposer-id can commit

an effective resolution. In Raft protocol , the unique node whose term is the biggest can be the leader. The distinction leader' s legitimacy strategy of Multi-paxos is the same as Raft. Whose proposer-id is larger, who is legal, and the proposer-id is the only. Therefore, in fact, at the same time, they only have a valid leader. In the raft protocol leader election algorithm, the newly elected leader already has all the logs that can be committed, while multi-paxos does not need to guarantee this. This also means that multi-paxos needs extra flow to get the logs that have already been committed from other nodes. Therefore, the raft protocol log can simply flow from the leader to the follower, while multi-paxos requires additional processes to complete the committed log.

2.4 Different log continuity

Raft protocol emphasizes the continuity of the log, but multi-paxos allows the log is not continuous. The continuity of the log contains such a nature: If two different nodes have the same sequence of logs, as long as the term is the same, the two logs must be the same, and it must be same with the previous log. This makes it convenient that when the leader wants the follower to synchronize the log. At the same time, log commit of Raft protocol is also continuous. If a log is committed, all the logs before the log are committed. If a log can be committed, all the logs before it can be committed. This also guarantees that when the Raft protocol is elected for the leader, a node in the majority must have all the committed logs. This is because that the last committed log is recorded by at least the majority. Due to the log continuity, having a log for the last commit means having all the commit logs, that is, at least one majority has all committed logs. For multi-paxos, it allows the log is not continuous. Each log need to be confirmed that if it can be individually committed. So when a new leader is generated, it has to reconfirm each uncommitted log to determine if they can be committed. Even the new leader may miss the logs that can be committed and need to get the missing committable logs from other nodes through the Paxos phase 1. So essentially, the two protocols are the same. If a log is owned by the majority, then it can be committed, but the leader needs to know this in some way. In the meantime, in order for the log that has been committed to not be overwritten by the new leader, the new leader needs to have all the logs that have been committed(or can be committed) to make it works normally. The difference between the two is the way that the leader confirms committing and gets all the logs that can be committed, and the difference is due to whether the logs are consistent or not. The Raft protocol uses log continuity to simplify the process.

Chapter 3

Mechanism of Raft

3.1 A brief discription of Raft

There are three types of roles in a cluster organized by the Raft protocol: Leader, Follower and candidate.

The state changes of this three types of character changes are as shown in figure 3.1.

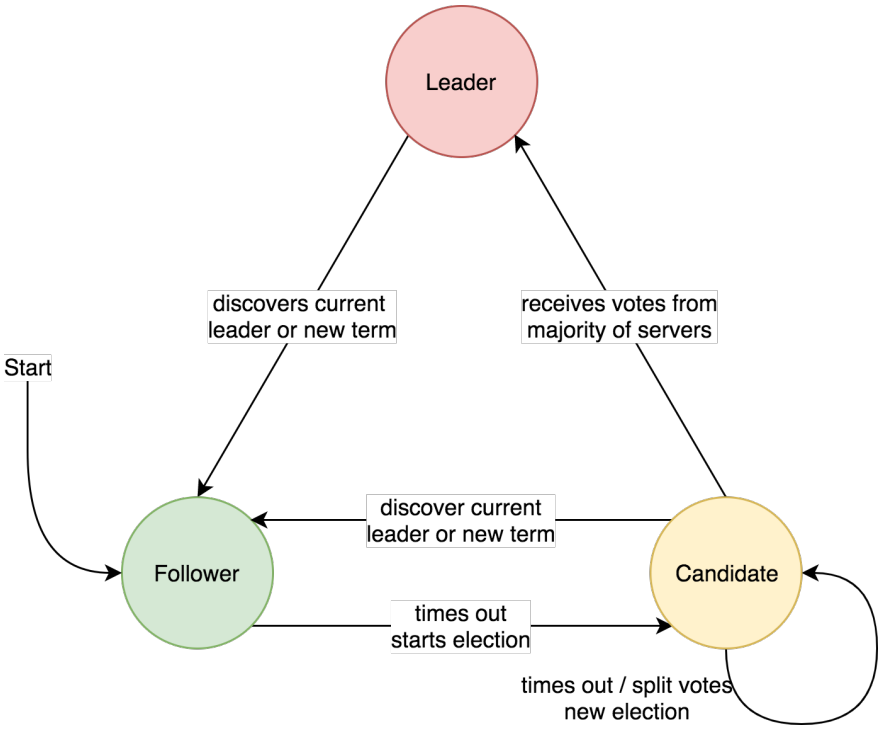

Figure 3.1: The state changes of them

3.2 The process of election

Let us give a simple example. As shown in figure 3.2, a minimum Raft democratization requires three participants (pictured below: A, B, C) so that a majority of votes exists. A, B and C are all followers by the initial state, and then launched one election of the three possible situations. The first two in the picture below can all be chosen as leader. The third one shows that this round of voting is invalid (Split Votes): each side has voted for itself. As a result, no one obtains the majority vote. After that, each participant randomly elected for the Election Timeout to re-initiate the voting until one party won the majority vote. The first from the timeout to resume the voting in the direction of the other two are still timeout request for a vote, then they can only vote for each other, and soon reached an agreement.

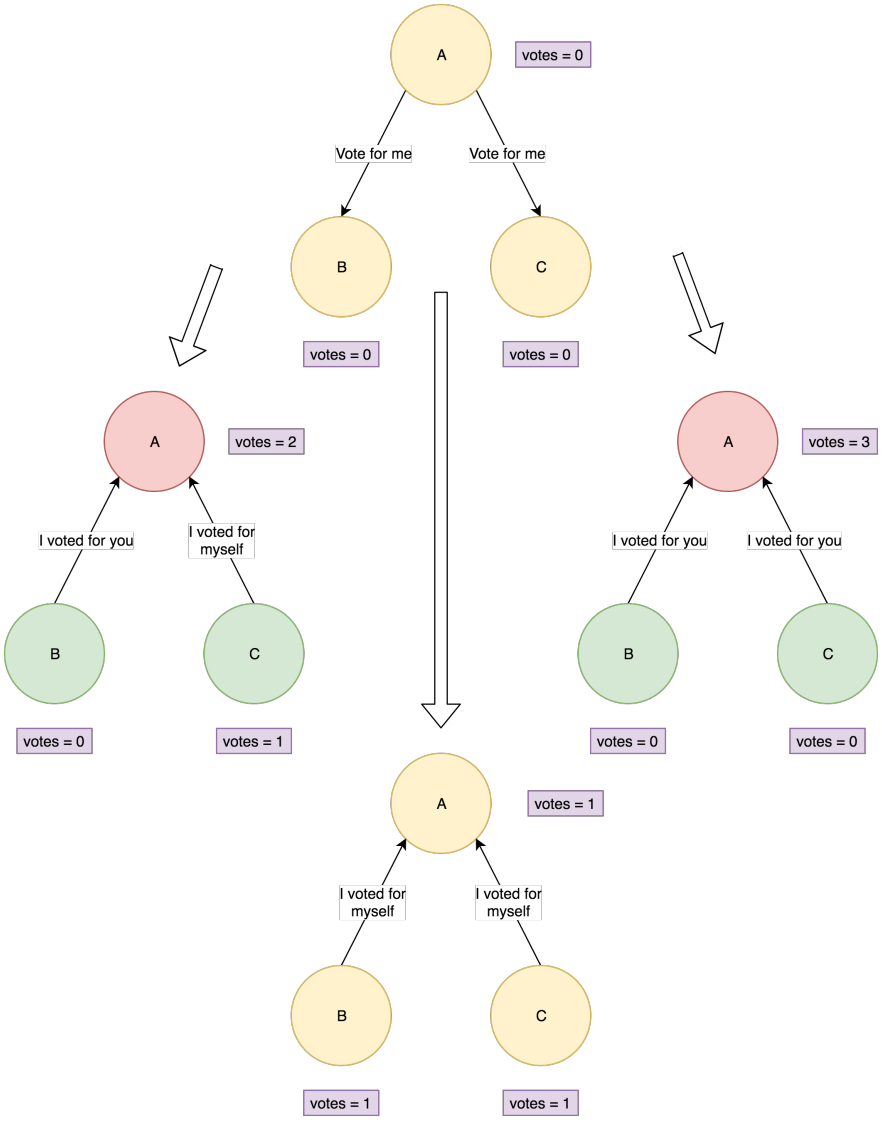

Figure 3.2: vote states

If a leader is selected, leader maintains its dominance by sending regular heartbeat messages to all followers. If a follower did not receive the leader's heartbeat for a period of time, it is assumed that the leader may have been crashed and started the election process again.

3.3 Impact of the leader node on consensus

Let's discuss the impact of leader nodes on consistency. The Raft protocol strongly depends on the availability of leader nodes to ensure the consistency of cluster data. As shown in figure 3.3, the flow of data can only be transferred from the leader node to the follower node. After the data is submitted to the cluster leader node, the data received by the leader node is in a uncommitted state. The Leader node then concurrently replication data to all the follower nodes and waits for the receiving response, ensuring that at least half of the nodes in the cluster have received after the data to the client to confirm the data has been received. Once the data is sent to the client to receive the ACK response, it indicates that the data status has been committed. The leader node then notifies the follower node that the data status has been submitted.

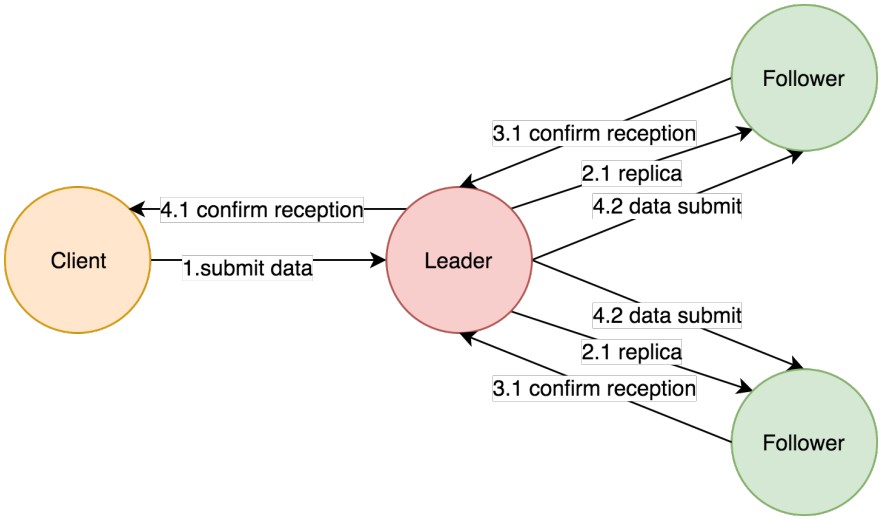

Figure 3.3: Total process diagram

During this process, the master node may crash at any stage. Let' s see how the Raft protocol guarantees data consistency for different stages.

3.4 The process of Raft

3.4.1 Before data arrives at the leader node

As shown in figure 3.4, leader crash does not affect consistency at this stage.

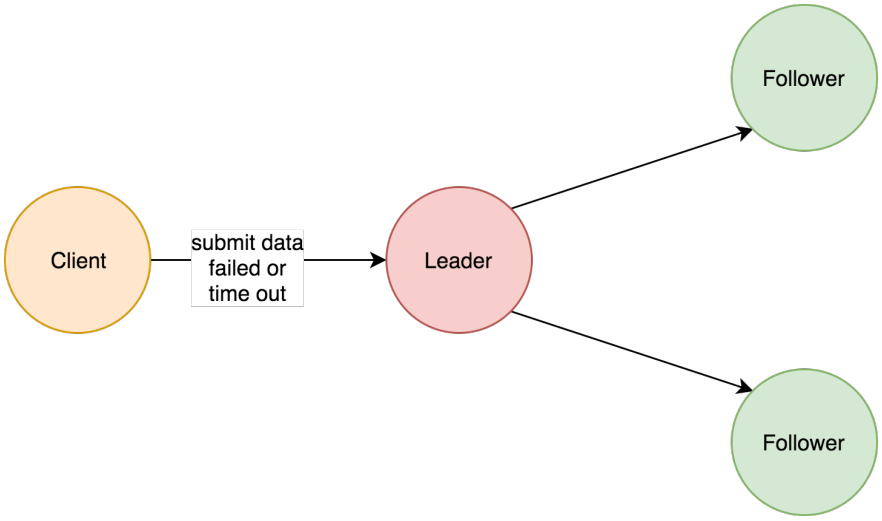

Figure 3.4: Process diagram

3.4.2 Data arrives at the leader node but not replicated to the follower node

As shown in figure 3.5, leader crashed at this stage, the data is uncommited state. Client will not receive Ack, it will think timeout failed, and can safely initiate a retry. Follower nodes do not have the data, after re-select the new leader, client retry to re-submit successfully. After the original leader node is recovered, it joins the cluster as a follower, synchronizes the data from the new leader of the current term, and enforces the consistency with the leader data.

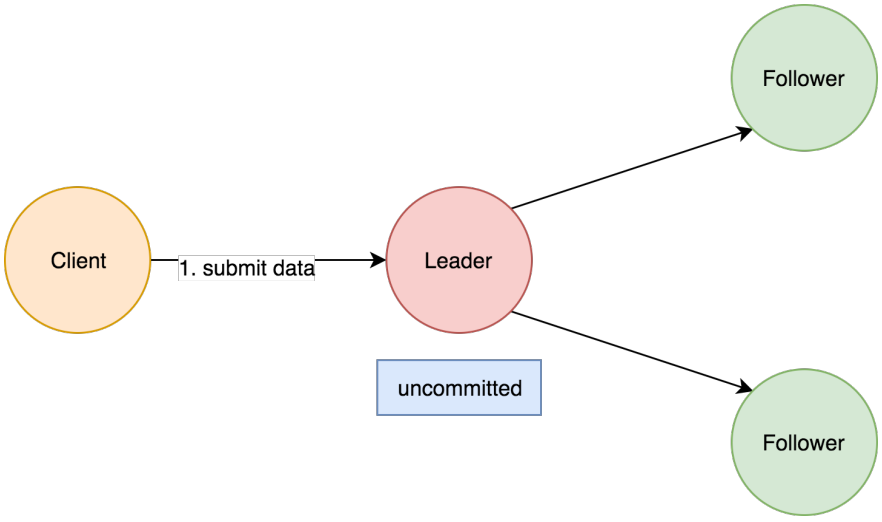

Figure 3.5: Process diagram

3.4.3 The data arrives at the leader node, successfully replicated to all follower nodes, but has not yet response to the leader

As shown in figure 3.6, leader crashed at this stage, although the data in the follower node is in the uncommitted state but consistent. the data can be committed after Re-electing the leader. At this time, client does not know whether it has been committed successfully, it can retry commit. In this case, Raft requires that RPC requests be idempotent, that is, to implement the internal deduplication mechanism.

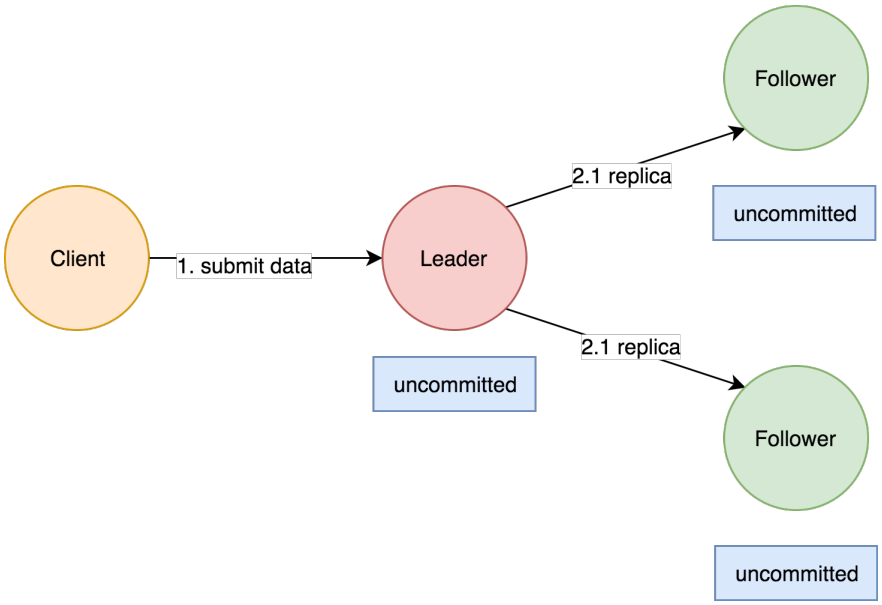

Figure 3.6: Process diagram

3.4.4 The data arrives at the leader node, successfully replicated to a part of follower nodes, but has not yet response to the leader

As shown in figure 3.7, leader crashed at this stage, the data in the follower node is in the uncommitted state and not consistent. The Raft protocol requires that votes be cast only to nodes that have the most recent data. So the node with the latest data will be selected as the leader and then forced to synchronize the data to follower, the data will not be lost and eventually consistent.

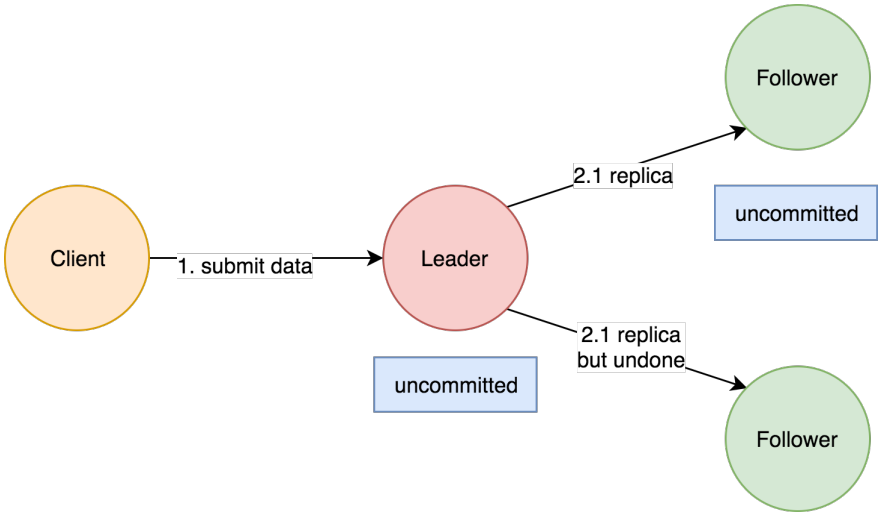

Figure 3.7: Process diagrams

3.4.5 The data arrives at the leader node, successfully replicated to all or most follower nodes, the data in the leader is in the committed state, but the follower is in the uncommitted state

As shown in figure 3.8, leader crashed at this stage. After re-electing the new leader, the process is the same as in Phase 3.

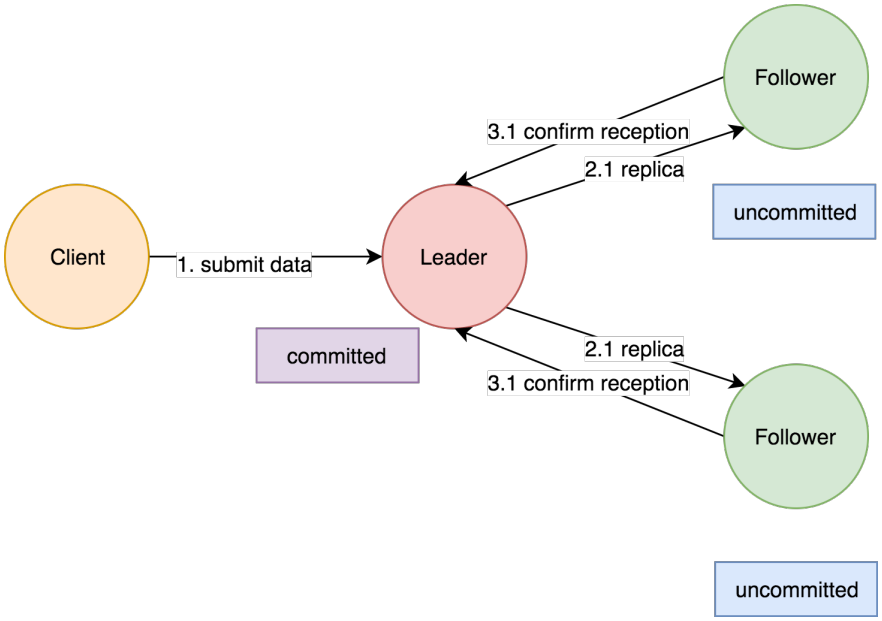

Figure 3.8: Process diagram

3.4.6 The data arrives at the leader node, successfully replicated to all or most follower nodes, the data is in the committed state at all nodes, but has not responsed to the client

As shown in figure 3.9, leader crashed at this stage. the internal data of the cluster is already consistent. Client repeat retries based on idempotent policies have no effect on consistency.

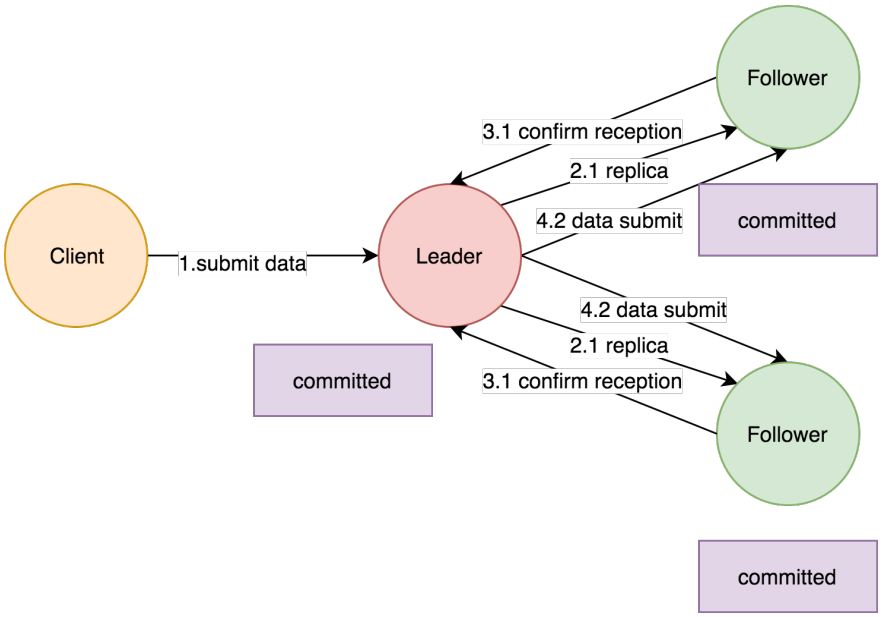

Figure 3.9: Process diagram

3.4.7 Network partition led to double leader situation

As shown in figure 3.10, the network partition separates the original leader node from the follower node. The follower can not receive the heartbeat of the leader and will initiate the election to generate a new leader. At this time it has produced double leader, the original leader is alone in a district, the data submitted to it is impossible to be replicated to most nodes, so it cannot be submitted successfully. Submitting data to the new leader can be successful. After the network is recovered, the old leader discovers that there is a new leader with a new term in the cluster and it will automatically downgrad to follower, and the data is synchronized with the new leader to achieve the same cluster data.

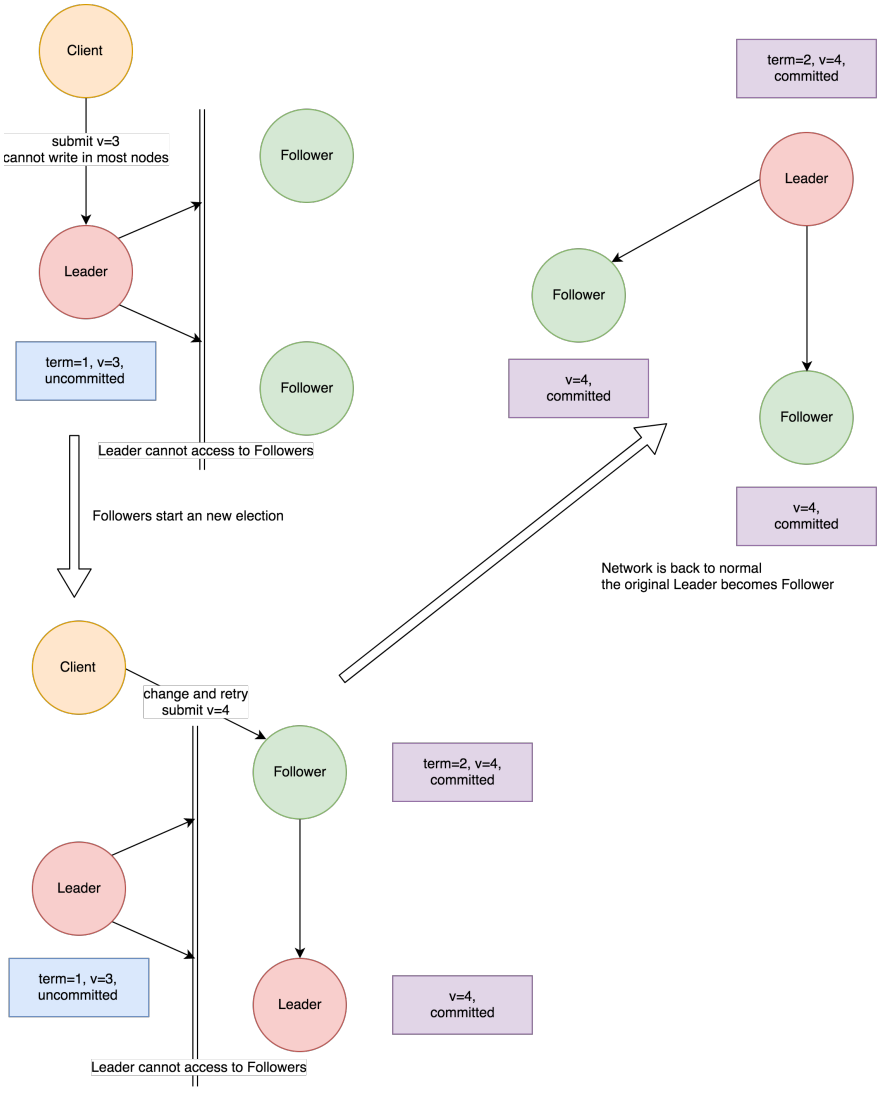

Figure 3.10: Process diagram

3.4.8 Summary

To summarize, we exhaustively analyzed all the situations faced by the smallest cluster (3 nodes). It can be seen that the Raft protocol can well cope with the consensus problem and is easy to understand.

Chapter 4

A Simple Implementation of Raft

4.1 System structure

As shown in figure 4.1, we used a system structure diagram to describe our finished work, it roughly describes how this system works.

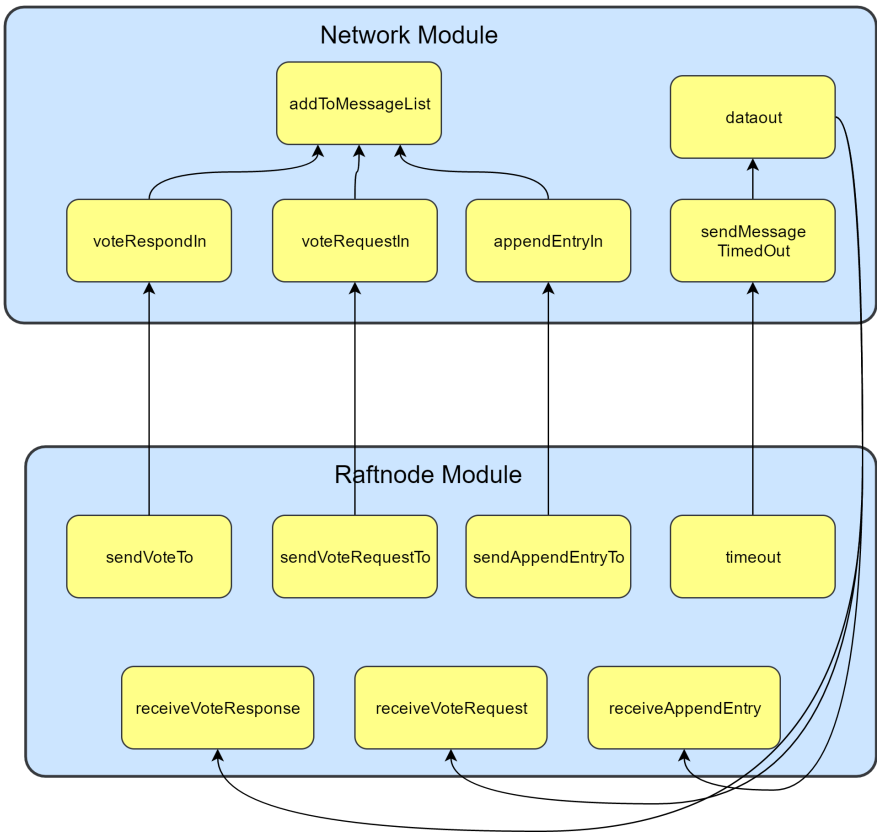

Figure 4.1: System structure diagram

4.2 Main functions & algorithms

We chose several important functions as an example, and listed the flow charts and specific algorithms for these functions.

4.2.1 startElection()

This function begins when there is a timeout or at the start. There is not a valid leader in the network. Through this function, the node start a election, clear the vote status, become a candidate, and sending vote request to other nodes.

Flow chart

The flow chart of this algorithm shown in figure 4.2.

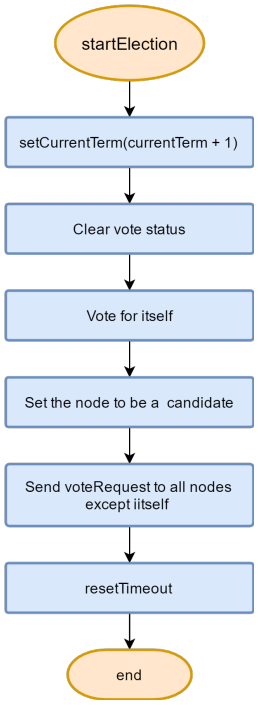

Figure 4.2: Start election

Algorithm

Algorithm 1: follower starts a election

1: **procedure** STARTELECTION
2: SETCURRENTTERM($currentTerm + 1$) ▷ update current term
3: **for** $node \leftarrow$ each node in network **do**
4: $node.votedForMe \leftarrow false$ ▷ clear vote status

```
5:    end for
6:    voteFor(id, TRUE)                                   ▷ vote for myself
7:    currentLeader ← invalid
8:    state ← stateCandidate
9:    for node = each node in network except myself do
10:       if node ≠ id then
11:           requestVoteFrom(node.id))
12:       end if
13:   end for
14:   resetTimeout                                        ▷ randomize election timeout to avoid forever loop
15: end procedure
```

4.2.2 receiveVoteRequest()

When a node receives a vote request, It will determine whether its own term is smaller than the requesting node, and if so, it will be a follower and reset timeout. Then, it will call the function shouldVote to determine vote or not. Finally, it will reset timeout.

Flow chart

The flow chart of this algorithm shown in figure 4.3.

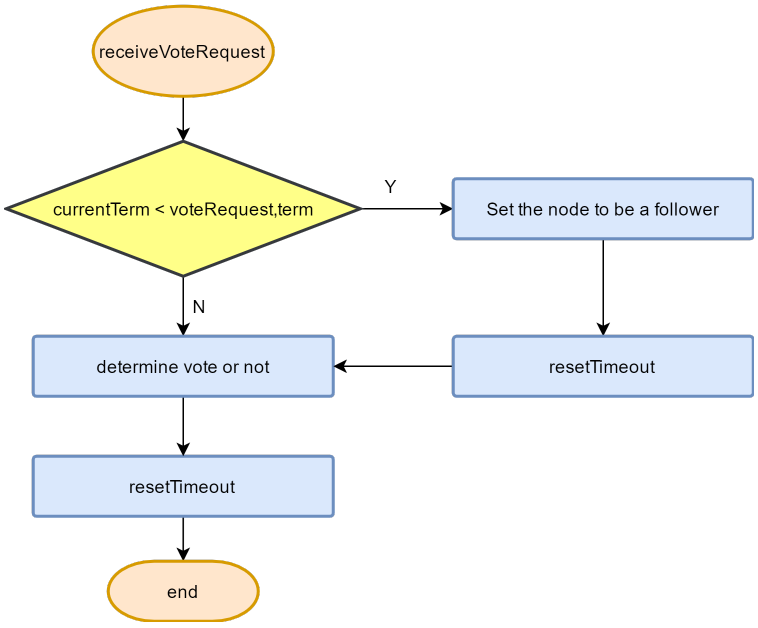

Figure 4.3: Receive vote request

Algorithm

Algorithm 2: node receives a vote request

```
1: procedure RECEIVEVOTEREQUEST(voteRequest)
2:     if currentTerm < voteRequest.term then   ▷ if my term < request node's term, become a follower unconditionally
3:         SETCURRENTTERM(voteRequest.term)
4:         state ← stateFollower
5:         RESETTIMEOUT
6:     end if
7:     if SHOULDVOTE(voteRequest) = TRUE then                ▷ determine vote or not
8:         VOTEFOR(voteRequest.id, TRUE)
9:         currentLeader ← invalid
10:        RESETTIMEOUT
11:    else
12:        VOTEFOR(voteRequest.id, FALSE)
13:    end if
14:    RESETTIMEOUT
15: end procedure
```

4.2.3 receiveVoteResponse()

When a node receives a vote response, it will finish the following process. Process 1, if it is not a candidate, end. Process 2, if its term is smaller than the response' s, it will become a follower and reset timeout, end. Process 3, if its term is bigger than the response' s, end. Process 4, if the response node is not in the network, ignore it, end. Process 5, if the response is not approval, end. Process 6, statistics own votes, if the totalvotes >= the amount of nodes/2+1, the current node will become a leader and reset timeout.

Flow chart

The flow chart of this algorithm shown in figure 4.4.

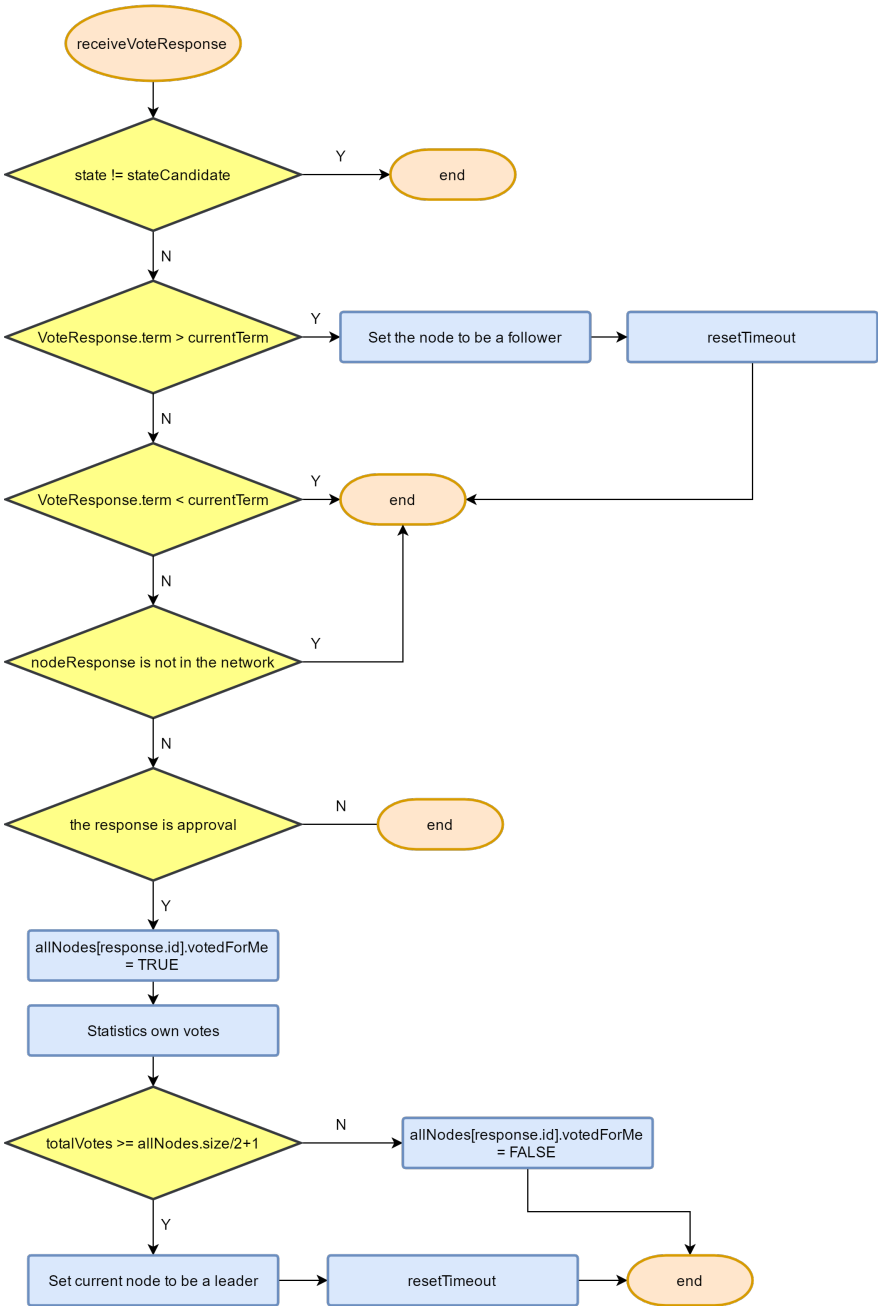

Figure 4.4: Receive vote response

Algorithm

Algorithm 3: node receives a vote response

1: **procedure** RECEIVEVOTERESPONSE(voteResponse)
2: **if** $state \neq stateCandidate$ **then**
3: return
4: **end if**

```
5:    if response.term > currentTerm then
6:        Set current node to be a follower
7:        resetTimeout
8:        return
9:    end if
10:   if voteResponse.term < currentTerm then                ▷ ignore outdated message
11:       return
12:   end if
13:   if allNodes.contains(voteResponse.id) = FALSE then      ▷ ignore unknown nodes
14:       return
15:   end if
16:   if voteResponse.approve then                            ▷ count votes if approved
17:       allNodes[response.id].votedForMe ← TRUE
18:       Statistics own votes
19:       if totalVotes >= allNodes.size/2 + 1 then           ▷ become leader if gets majority votes
20:           Set current node to be a leader
21:           resetTimeout
22:       else
23:           allNodes[response.id].votedForMe ← FALSE
24:       end if
25:   end if
26: end procedure
```

4.3 Test

4.3.1 Develop environment

operating system	Windows
translater	MingW gcc 7.2.0
Development Platform	Qt

4.3.2 Test steps

Three-nodes situation

- Start simulate, node 2 has been elected as a leader, as shown in figure 4.5. The number of candidate stand for the votes of it. Only when a node' s state has changed or votes has increased, the simulate program will display a new row.

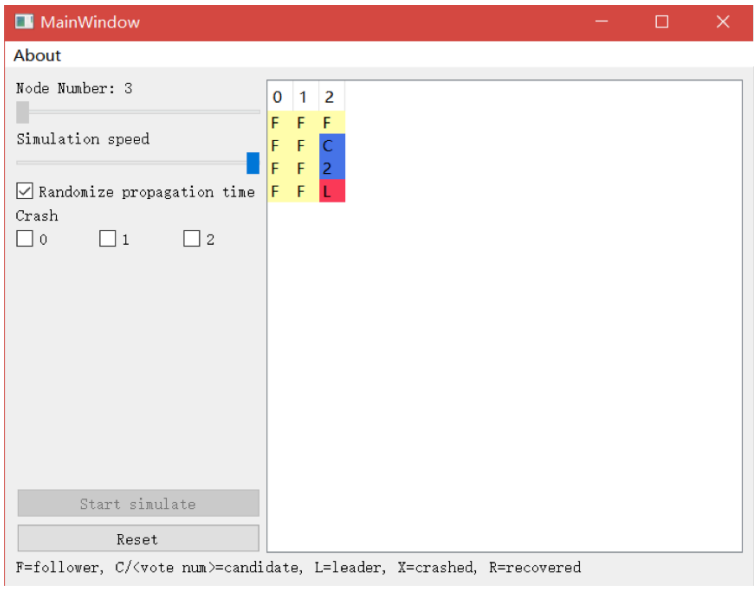

Figure 4.5: test of three nodes

- Node 2 crashed, and then node 1 has been elected as a new leader, as shown in figure 4.6.

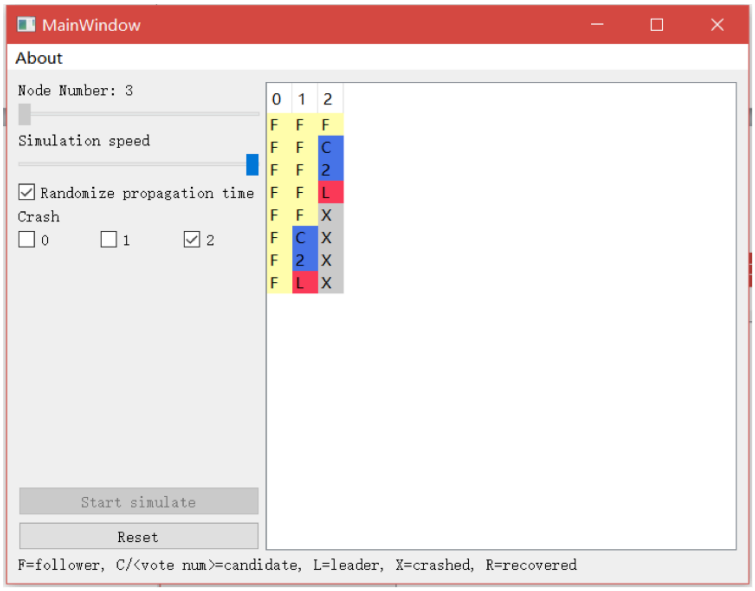

Figure 4.6: test of three nodes

More-nodes situation

- Node 3 has been elected as a leader, as shown in figure 4.7.

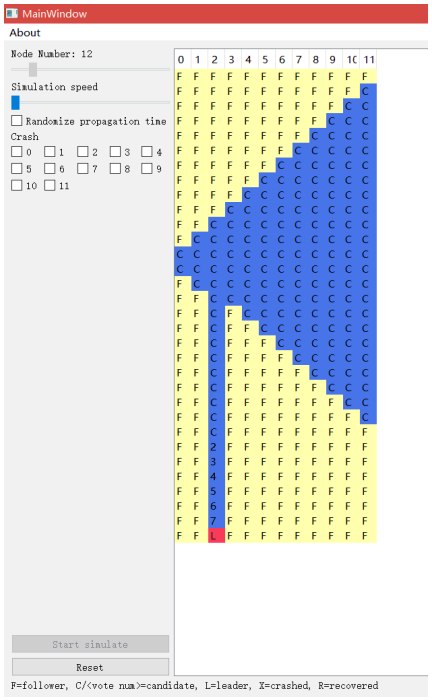

Figure 4.7: test of more nodes

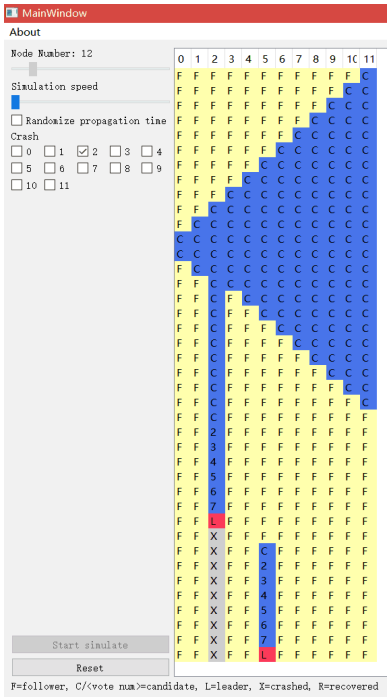

Figure 4.8: test of more nodes

- Node 3 crashed, and then node 6 has been elected as a new leader, as shown in figure 4.8.
- As shown in figure 4.9 and figure 4.10, this is a more complex case.

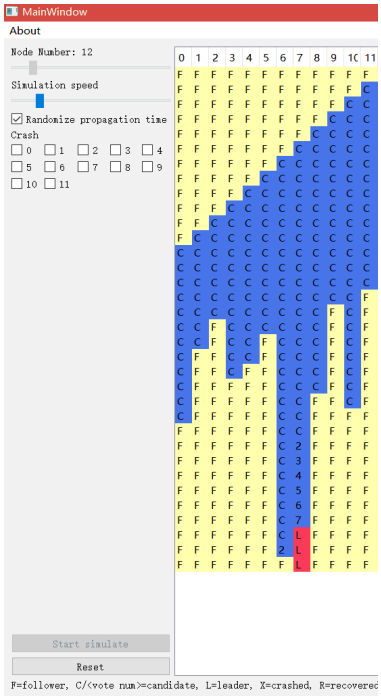

Figure 4.9: test of more nodes

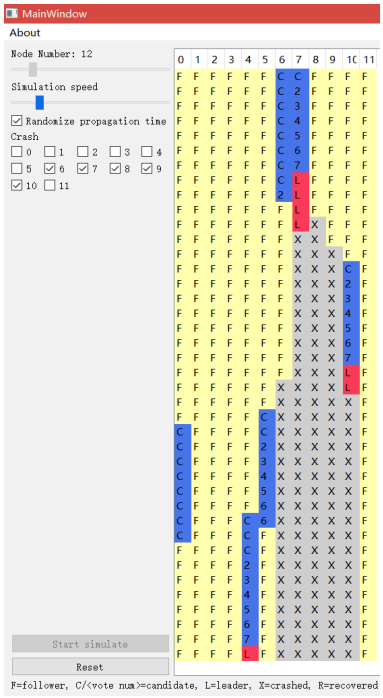

Figure 4.10: test of more nodes

Appendix A

References

[1] Ongaro, D., & Ousterhout, J. K. (2014, June). In search of an understandable consensus algorithm. In USENIX Annual Technical Conference (pp. 305-319).

[2] Ongaro, D., & Ousterhout, J. (2015). Raft consensus algorithm.

[3] Howard, H., Schwarzkopf, M., Madhavapeddy, A., & Crowcroft, J. (2015). Raft refloated: do we have consensus?. ACM SIGOPS Operating Systems Review, 49(1), 12-21.

[4] 张晨东, 郭进伟, 刘柏众, 储佳佳, 周敏奇, & 钱卫宁. (2015). 基于 Raft 一致性协议的高可用性实现. 华东师范大学学报: 自然科学版, (5), 172-184.

[5] 鲁子元. (2017). 浅析 RAFT 分布式算法. 信息技术, 41(9), 168-170.

Appendix B

Division of Work

- Sixu Hu : protocol, simulatorm and GUI implementation.
- Yifu Deng : resource integration and document writing.
- Junhua Zhang : major part of document and report conposition.