École Polytechnique Fédérale de Lausanne

BFT Baxos: Robust and Efficient BFT Consensus using Random Backoff

by Zhanbo Cui

Master Project Report

Prof. Bryan Alexander Ford
Project Advisor

Pasindu Nivanthaka Tennage
Project Supervisor

EPFL IC DEDIS
January 6, 2023

Chapter 1

Introduction

Consensus protocol can ensure distributed system nodes agree on a unique value from their proposed values if all the nodes follow the protocol correctly. It has four properties: validity, integrity, agreement and termination[1]. However, some nodes might not follow the protocol. They might omit to send messages or send contradicting messages to other nodes. If a node behaves arbitrarily, it is called a byzantine node. BFT consensus protocol should allow all correct nodes to reach a common decision despite the byzantine fault.

For liveness reasons in consensus protocol, most BFT consensus protocols is leader-based. For instance, PBFT[2] use a fixed leader to avoid contention to guarantee liveness and Hotstuff[4] use a rotating leader to avoid contention to guarantee liveness. However, leader-based consensus might still be a risk of liveness loss in the public network. The adversary can carefully analyze the network traffic and know who the leader node is, due to the leader node has massive message exchange with other nodes in the cluster. The adversary can perform DDoS attack on the leader node to delay its performance. Since the nodes of cluster always use a timer to measure if the leader node is alive, the delayed correct leader might be demoted and lose the ability to coordinate the consensus process. So the leader-based consensus might not be able to guarantee progress in the public network where an adversary exists[3].

BFT Baxos is a leader-less BFT consensus inspired by random backoff in CSMA. We use random exponential backoff to avoid contention to guarantee liveness and use 3 phases and quorum certificate to guarantee safety. We use Byzantine Paxos as the baseline and make two novel contributions to this project. Aadapt Byzantine Paxos with quorum certificate to decrease the complexity of communication. Meanwhile, adapt Byzantine Paxos with a random exponential backoff mechanism as a leader replacement. This work is never explored before.

The rest of the report is organized as follows. Chapter II presents BFT Baxos algorithm design, pseudocode, the proof of safety and liveness. Chapter III describes our Golang implementation of BFT Baxos and the evaluation. Chapter IV discuss the future work.

Chapter 2

Design

2.1 System Model

We assume the network is partially-synchronous, An unknown GST exists such that once the GST is reached, there is an upper bound on message transmission from one node to another node.

We can tolerate the most f byzantine fault node within a cluster of n = 3f+1 nodes, since our BFT Baxos is based on Byzantine Paxos.

We use threshold signature to form quorum certificate, so we assume it can only be combined when having majority (2f+1) valid partial signature. f malicious nodes can not combine to a valid quorum certificate.

2.2 The BFT Baxos Algorithm

BFT Baxos is three phases consensus as Byzantine Paxos to achieve Byzantine fault tolerance. But the difference with Byzantine Paxos, which requires using two broadcasts to ensure safety properties, BFT Baxos leverages quorum certificates to convince Acceptors, and only the Proposer needs to broadcast. This way can reduce the complexity of communication and guarantee safety properties at the same time. As a leader-less consensus, every BFT Baxos node can propose freely, but we need to handle contention when more than one proposer propose at the same time. Inspired from random backoff in CSMA. We ask the proposer back off for a random time to prevent further collisions if Acceptors notice there are concurrent requests for the same consensus instance. Try to achieve every node can have the fair share for the shared resource and improve the resilience from changing network delays. Our REB mechanism guarantee eventually there is one Proposer successfully commit his value.

2.3 Safety in BFT Baxos

The following is a description of how the three phases BFT Baxos works to guarantee safety:

a)**Prepare - Promise**: A node who wants to propose a new value initiate consensus by broadcasting a *Prepare message* with a ballot number to all Acceptors. The Acceptors send a *Promise*

message to the Proposer, if they have not promised any *Prepare message* with a higher or equal ballot number than the ballot number received in the *Prepare message*. If Acceptors has previously accepted any value, they need to piggyback this value and the corresponding ballot number for this value in the *Promise message*. Considering that we need to implement Byzantine fault tolerance, to convince the Proposer, Acceptors also need to attach a PreAccept quorum certificate to prove the value they accepted is safe, we denote it as *preAcceptQC*. After the Proposer collecting *Promise messages* from a majority (2f + 1 or more), Proposer can get a set of *Promise message*, we denote this set as *Promise set*. If all *Promise messages* within *Promise set* indicate that there is no previously accepted value, then the Proposer can select its new value as *Prepropose value*, otherwise it should select the previously accepted value which has the highest ballot number from the *Promise set* as *Prepropose value*.

b)**Prepropose - Preaccept**: In this phase, the Proposer who successfully forms a *Promise set* can broadcast the *preProposeValue* with the same ballot number used in previous phase. The *Prepropose message* need to contain the *Promise set* to convince Acceptors that the *Prepropose value* is selected properly according to the *Promise set*. For an Acceptor accepts a *Prepropose message*, firstly, the ballot number in the *Prepropose message* is equal or greater than the ballot number that it previously promised. Secondly, the *Promise set* should be valid which should satisfy two conditions: 1. the size of set should equal or greater than the majority; 2. the *Promise messages* within *Promise set* are valid. Finally, the *Prepropose value* should be the value selected according to the *Promise set*. Upon accepting a valid *Prepropose message* from the Proposer, Acceptors update their *preaccept_ballot number*, *preaccept_value* and generate a partial certificate by signing on the tuple <preaccept_ballot, preaccept_value>. The *Preaccept message* sent to the Proposer piggyback this partial certificate. The Proposer, upon receiving valid *Preaccept message* from a majority of Acceptors, can combines the *preAcceptQC*. The *preAcceptQC* represents there is a majority of Acceptors have preaccepted the value preproposed by the Proposer.

c)**Propose - Accept**: In this phase, the Proposer who successfully combines a *preAcceptQC* can broadcast the *proposeValue* with the same ballot number used in previous phase. The *Propose message* need to contain the successfully combined *preAcceptQC*. For an Acceptor accepts a *Propose message*, firstly, the ballot number in the *Propose message* is equal or greater than the ballot number that it previously promised. Secondly, *preAcceptQC* in the *Propose message* should be valid. Finally, the tuple <propose_ballot, propose_value> in the *Propose message* should equal to the tuple <preaccept_ballot, preaccept_value> which signed by a majority, the corresponding multi-signature be placed in the *preAcceptQC* and can be verified. Upon accepting a valid *Propose message* from the Proposer, Acceptors update their accept_ballot number, accept_value and *preAcceptQC*, and generate a partial certificate by signing on the tuple <accept_ballot, accept_value>. The *Accept message* sent to the Proposer piggyback this partial certificate. The Proposer, upon receiving valid *Accept message* from a majority of Acceptors, can combines the Accept quorum certificate, we denote it as *acceptQC*. The *acceptQC* represents there is a majority of Acceptors have accepted the value proposed by the Proposer. Finally, Proposer can safely commit the value it proposed and broadcast the *Commit message* which contains the *acceptQC* to inform Acceptors commit previously accepted value safely.

To summarize, BFT Baxos leverage Promise set and two quorum certificates to ensure safety

properties in the Byzantine environment. Since BFT Baxos does not have the broadcast phase, the Promise set exists to prevent malicious proposers from broadcasting contradictory Prepropose messages to Acceptors. Meanwhile, The second phase exists to help Acceptors can provide a proof(quorum certificate) in the first phase to convince the Proposer the value they previously accepted is safe.

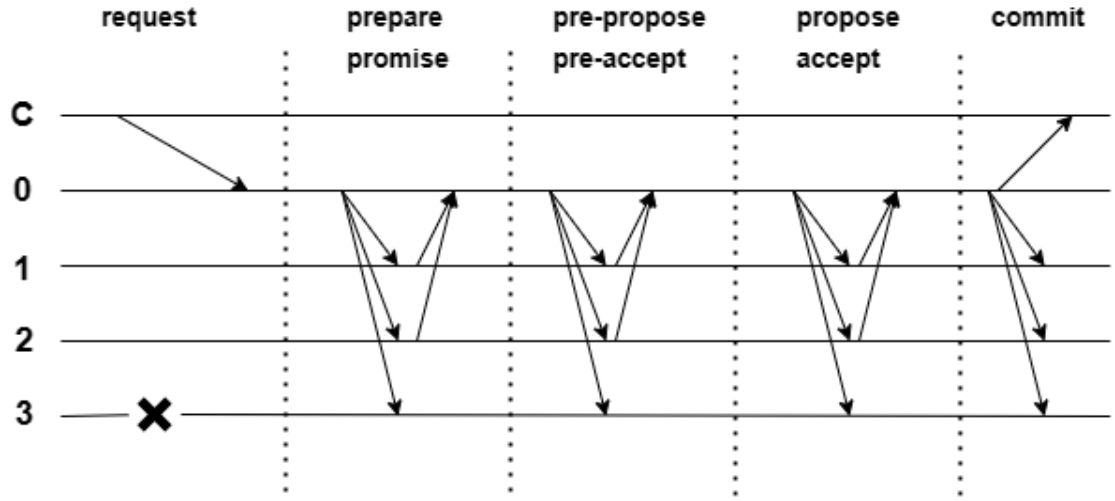

Figure 2.1: Three phases in BFT Baxos

2.4 Safety Proof in BFT Baxos

We now present the proof of safety properties in consensus for single instance BFT Baxos:

Validity: In BFT Baxos. For an Acceptor to decide on a value v, it should receive a *commitValue* and an *acceptQC* associated with this value from the Proposer. The Acceptor can verify whether the *commitValue* is valid through *acceptQC*. The proposer can combine a valid *acceptQC* only if it receives at least 2f + 1 number of *Accept message* containing valid partial signatures from honest Acceptors. This *acceptQC* can prove quorum honest nodes have accepted the *proposeValue*. For the Acceptors to send the *Accept message*, they should have received the *proposeValue* attached with a *preAcceptQC*. The *preAcceptQC* can prove the *proposeValue* is a valid value, so the honest nodes can accept *proposeValue* safely. As the same as the *acceptQC*. The Proposer can combine a valid *preAcceptQC* only if it receives at least 2f + 1 number of *Preaccept messages* containing valid partial signatures from honest Acceptors. For the Acceptors to send the *Preaccept messages*, they should have received the *preProposeValue* and a *Promise set* whose size is at least 2f + 1. This set can prove the validity of *preProposeValue*. As for the proposer to formulate a valid *Promise set*, it should have received at least 2f + 1 *Promise messages* from Acceptors. Acceptors indicate their previous *preaccept_value* and attach a *preAcceptQC* to prove this value is valid. There are two cases to consider in this case; 1) All *preaccept_value* and *preAcceptQC* are nil in the *Promise messages* so that the Proposer can select his proposal in the *Prepropose message* or 2) the *preProposeValue* is the value that has the highest ballot number in the *Promise messages*.

In the first case, the *preProposeValue* is the value proposed by this Proposer, so this value is valid, hence the decided value is also proposed by this Proposer. In the second case, the Proposer selects a unique value that was previously accepted by one or many Acceptors. Since there is a *preAcceptQC* attached to this *preProposeValue*, so this *preProposeValue* is valid. We can use the same argumentation mentioned before to trace this value should be proposed by a previous Proposer with a smaller ballot number. Hence, the committed value is also the value proposed by a Proposer. Hence the validity property holds.

Agreement: To prove the agreement property, it is sufficient to prove that if an Acceptor A has accepted value *v* in ballot number *i*, then no value *v' != v* can be decided in any smaller ballot number. We can use induction to prove it.

Assume that Acceptor *A* has accepted this value *v* in ballot number *i*. for this to happen, *A* must have received a *Propose message* from a Proposer. The Proposer must attach a *preAcceptQC* with this *Propose message*, this *preAcceptQC* ensures the *v* in the *Propose message* is preaccepted by quorum size Acceptors. And the condition for a Acceptor to preaccept *v* is receiving a *preproposeValue v* and a *Promise set*; meanwhile the *preAcceptQC*s in the *Promise set* indicate two cases: 1) no value preaccepted before, then the agreement property is proved, because there is no value different from *v* that is preaccepted/accepted in a smaller ballot number, hence there is no value will be decided differently from *v* in any smaller ballot number. 2) a value *v* with ballot number *j* is preaccepted by quorum Acceptors(*j* is smaller than *i* but is the highest ballot number collected in the *Promise set*). There is no such previous ballot number *j* for *i* when *i = 0* since we start to propose at ballot number 0. In turn, when *i != 0*, there is no ballot number *k* where *j > k > i* could make another *preproposeValue v'* be accepted. Since if value *v* has been preaccepted by quorum Acceptors at ballot number *j*, there is always an intersection of Acceptor sending the *preAcceptQC* of *v* at ballot number *j+1* to formulate the *Promise set*, so the *preproposeValue* will be *v* at ballot number *j+1*. And so on for *j+2,...k-1,k*, the *preproposeValue* will always be *v*. Hence there is impossible for Acceptors to preaccept another *v'*, hence the Proposer can't combine a new *preAcceptQC* of *v'* through the partial signatures, hence the value *v'* will never be accepted. By using the inductive hypothesis on ballot number *j* to deduce there is no value *v' != v* can be accepted in any ballot number smaller than *j* (ballot numbers *0, . . . , j - 1*). This proves agreement.

Integrity: The integrity property holds in single-choice BFT Baxos. There is a boolean variable decided which is updated just once from false to true when receiving the *Commit message* at the side of the Acceptor and broadcasting the *Commit message* at the side of Proposer. So each node only decides the value once.

2.5 Liveness challenges in BFT Baxos

We leverage *random exponential backoff (REB)* to guarantee the termination when there are multiple competing proposals from different proposers. The proof of termination has been proved in the Baxos paper[3]. Since BFT Baxos is a three phases consensus, so we need to modify

the formula used to calculate backoff time in Baxos as following:

$$k * 3^l * RTT$$

Where $k \in (0,1) \in Q$ and l is the number of retries. We focus on the liveness chanllenges from the Byzantine environment since malicous node might not generate k randomly and increase l, ballot number monotonically, however our proof of termination is based on the Proposer must backoff the time caculated from fomula honestly when contention happened. We leverage VRF, quorum certificate, retry table, Byzantine-tolerant timestamp and backoff time check mechanism to ensure the message proposed by the Proposer is "random backoff" under the view of Acceptors.

2.5.1 Generate *k* randomly

we use the verifiable random function(VRF) to guarantee k is generated randomly and can be verified. Proposer can use private key and the common seed to generate random number and use the public key to generate *k_proof* for this random number. Proposer should use this k to calculate the backoff time. When it complete the backoff and retry the Prepare-Promise phase, this proposer should attach k and *k_proof* within *Promise message*. On the Acceptors side, they can use the corresponding public key, the common seed and the proof to verify if the number is generated randomly. Common seed at here can be a incremental sequence recognized by all the node in the cluster, but in order to make the generation of k more unpredictable and unmanipulable, we use the Byzantine-tolerant timestamp as the common seed, meanwhile, this timestamp is also usefull in our backoff check mechanism. This idea is firstly proposed in *Byzantine Ordered Consensus*[5] to guarantee the order commands in BFT SMR in a way that respects a natural extension of linearizability. In our BFT Baxos, we don't use it to coordinate the order of consensus instances, but as common seed in VRF and use it to achieve backoff check mechanism. An intuitive understanding of Byzantine-tolerant timestamp is that the median timestamp within a set of 2f + 1 timestamps is might not come from the honest node, but it must locate in the reasonable time interval. As for the seed of VRF, even if this timestamp come from malicious node, it doesn't matter for generating random k randomly. We will explain how to use byzantine-tolerant timestamp to achieve backoff check mechanism in the later section.

2.5.2 Increase *l* monotonically

Since our network is partially-synchronous, we can transfer the responsibility for counting the number of retries from the Proposer to the Acceptors. Each node in the cluster maintains a table to record the number of retries for the other nodes. Each time a retry Prepare message is received, the Acceptors increment the corresponding Proposer the number of retries in the table. When the Proposer successfully commits the value, the Acceptors will also clear the number of retries accordingly.

2.5.3 Increase *ballot number* monotonically

To achieve the monotonic increment of ballot number, we use the quorum certificate to restrict the proposer from arbitrarily increasing its own ballot number. Acceptors always piggyback a partial signature for *next ballot number* in the Prepare-Promise phase. So if Proposers don't start with ballot number 0, in other word, they need to attach with a *nextBallotQC* to prove the validity of their ballot number when they retry. There are many strategies on how to determine the next ballot number, the naive way is increasing only one on the previous ballot number.

2.5.4 Backoff check mechanism

Lastly, we need help honest Acceptors to know whether the Proposer indeed back off. When the Proposer retry, it should include the timestamp set collected in the previous Prepare-Promise phase in the new *Prepare message*, on the side of Acceptors, they can use receiving time minus median timestamp within timestamp set to calculate out a time duration. This duration should greater or equal to the backoff time calculated from the REB formula, if not, the Acceptors put the message into a waiting buffer and wait for the time is reached. We ensure that the termination property still hold in the Byzantine environment by ensuring that requests from malicious Proposer are deferred through the backoff check mechanism.

$$receiving\_time - median\_timestamp \geq k * 3^l * RTT + RTT$$

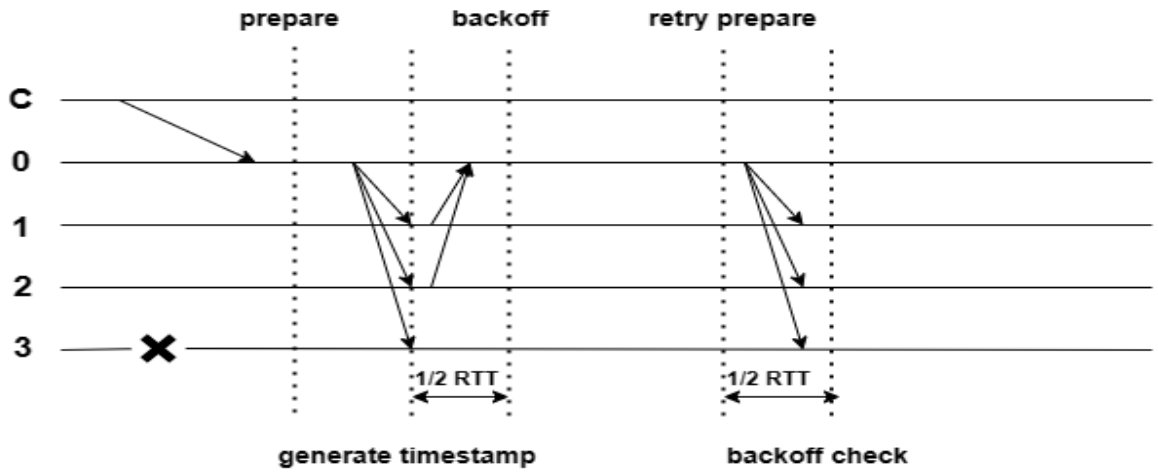

Figure 2.2: Random backoff check machenism

2.6 Pseudocode for BFT Baxos

Algorithm 1: BFT Baxos Prepare-Promise phase

```
1  Initialization of local variable:
2  retries := 0;
3  curBallot, promiseBallot, preAcceptBallot, acceptBallot := -1;
4  value, preAcceptValue, acceptValue, comiteValue := null;
5  preAcceptQC, acceptQC := null;
6  comitted := false;
7  promiseSet, timeStampSet, preAcceptSet, acceptSet := {};
8  k, kProof := nil;
9  retryTable := Map{};
10 nodeId;
11 Timer;
12 Proposer: onPropose(client_v)
13     curBallot += 1;
14     promiseBallot = curBallot;
15     value = client_v;
16     BroadCast Msg(PREPARE, promiseBallot, k, kProof, promiseSet, timeStampSet);
17     promiseSet = {};
18     timeStampSet = {};
19     Timer.start();
20 End;
21 All nodes: onMessage(PREPARE, prepareBallot, k, kProof, timeStampSet)
22     if ! isBackoff then
23         Wait for the backoff time
24     end
25     retryTable[proposer] += 1;
26     if promiseBallot < prepareBallot then
27         promiseBallot = prepareBallot;
28         contention = false;
29     else
30         contention = true;
31     end
32     timeStamp = Timer.Now();
33     Unicast Msg(PROMISE, contention, promiseBallot, acceptBallot, acceptValue,
         preAcceptQC, timeStamp);
34 End;
```

Algorithm 2: BFT Baxos Prepropose-Preaccept phase

```
1  Proposer: onMessage(PROMISE, contention, ballot, lastAcceptBallot, lastAcceptValue,
     lastPreAcceptQC, timeStamp) onCondition: promiseBallot == ballot
2      timeStampSet = timeStampSet ∪ timeStamp;
3      if isValidPromise AND !contention then
4          promiseSet = promiseSet ∪ PROMISE
5      end
6  End;
7  Proposer: onEvent( |promiseSet| >= 2f + 1)
8      Timer.cancel();
9      highestBallot, highestValue = HIGHESTBYBALLOT(promiseSet);
10     if highestValue != null then
11         preProposeValue = highestValue;
12     else
13         preProposeValue = value;
14     end
15     preProposedBallot = promiseBallot;
16     BroadCast Msg(PREPROPOSE, preProposedBallot, preProposeValue, promiseSet);
17     promiseSet = {};
18     Timer.start();
19 All nodes: onMessage(PREPROPOSE, preProposedBallot, preProposeValue, promiseSet)
20     if promiseBallot =< preProposedBallot then
21         contention = false;
22         if ! preProposeValueIsValidInPromiseSet then
23             return;
24         end
25         preAcceptBallot = preProposedBallot;
26         preAcceptValue = preProposedValue;
27         preAcceptSig = tsign(PREACCEPT, preAcceptBallot, preAcceptValue);
28         timeStamp = Timer.Now();
29         Unicast Msg(PREACCEPT, contention, preAcceptBallot, preAcceptValue,
             preAcceptSig, timeStamp);
30     else
31         contention = true;
32         timeStamp = Timer.Now();
33         Unicast Msg(PREACCEPT, contention, preAcceptBallot, preAcceptValue,
             preAcceptSig, timeStamp);
34     end
35 End;
```

Algorithm 3: BFT Baxos Propose-Accept phase

```
1  Proposer: onMessage(PREACCEPT, contention, ballot, preAcceptValue, preAcceptSig,
     timeStamp) onCondition: preAcceptBallot == ballot
2      timeStampSet = timeStampSet ∪ timeStamp;
3      if isValidSig AND !contention then
4          preAcceptSet = preAcceptSet ∪ preAcceptSig
5      end
6  End;
7  Proposer: onEvent( |preAcceptSet| >= 2f + 1)
8      Timer.cancel();
9      preAcceptQC = tcombine(<PREACCEPT, preAcceptBallot, preAcceptValue>,
         preAcceptSig ∈ preAcceptSet );
10     proposeBallot = preAcceptBallot;
11     proposeValue = preAcceptValue;
12     BroadCast Msg(PROPOSE, proposeBallot, proposeValue, preAcceptQC);
13     Timer.start();
14 All nodes: onMessage(PROPOSE, proposeBallot, proposeValue, preAcceptQC)
15     if promiseBallot =< proposeBallot then
16         contention = false;
17         if ! isValid(preAcceptQC) then
18             return;
19         end
20         acceptBallot = proposeBallot;
21         acceptValue = proposeBallot;
22         acceptSig = tsign(ACCEPT, acceptBallot, acceptValue);
23         timeStamp = Timer.Now();
24         Unicast Msg(ACCEPT, contention, acceptBallot, acceptValue, acceptSig,
             timeStamp);
25     else
26         contention = true;
27         timeStamp = Timer.Now();
28         Unicast Msg(ACCEPT, contention, acceptBallot, acceptValue, acceptSig,
             timeStamp);
29     end
30 End;
```

Algorithm 4: BFT Baxos Commit phase

```
1  Proposer: onMessage(ACCEPT, contention, ballot, acceptValue, acceptSig,
     timeStamp) onCondition: acceptBallot == ballot
2      timeStampSet = timeStampSet ∪ timeStamp;
3      if isValidSig AND !contention then
4          acceptSet = acceptSet ∪ acceptSig
5      end
6  End;
7  Proposer: onEvent( |acceptSet| >= 2f + 1)
8      Timer.cancel();
9      acceptQC = tcombine(<ACCEPT, acceptBallot, acceptValue>, acceptSig ∈ acceptSet );
10     BroadCast Msg(COMMIT, acceptBallot, acceptValue, acceptQC);
11 All nodes: onMessage(COMMIT, commitBallot, commitValue, acceptQC)
12     if ! isValid(acceptQC) AND !comitted then
13         comitted = true;
14         commitValue = acceptValue;
15     end
16 End;
```

Algorithm 5: BFT Baxos Utility

```
1  Proposer: onEvent(TIMEOUT)
2      cleanup(promiseSet, preAcceptSet, acceptSet);
3      retries += 1;
4      k, kProof = Random-Backoff(retries, timeStampSet);
5  End;
```

Chapter 3

Implementation and Evaluation

3.1 Implementation

We implemented BFT Baxos using Golang language. This project focuses on implementing the single instance version of BFT Baxos. We used Protobuf for serialization and deserialization and employed gRPC for inter-node communication. As for the crypto part, we used ed25519 packages to implement signature and quorum certificate, meanwhile the VRF library we used is also supported by ed25519 packages, since we can use the same set of public-private key pairs to achieve VRF and quorum certificate function in BFT Baxos. In practice, we assume there is a third party key distribution to responsible for key distribution, but in this project, we pre-generate the key pairs for each node, and each node can read the information of public key of each node in the cluster and the information of their own private key as they boot. We use the standard *time* library to returns a Unix time, the number of seconds elapsed since January 1, 1970 UTC as timestamp.

After the daemon process of node boot successfully, the node listen to different local ports which is defined in configuration files. We also implemented a simple client side by using cobra library. Users can connect to and send request to different nodes via client side command, this also means different users could initiate the consensus process from different node concurrently.

3.2 Evaluation

This section evaluates the time to reach a single BFT Baxos instance in clusters of varying numbers of nodes.

3.2.1 Experimental Setup

We designed three sets of tests, starting 4, 7 and 10 local nodes respectively, corresponding to the cluster can tolerate 1, 2 and 3 malicious nodes. Then we send the consensus request by using the client-side command to one of the node in the cluster, after waiting for the consensus to be reached, we calculate the time it takes from the initiation of the request to receiving the reply of the request. Each set of tests initiates ten requests and calculates the average time. All nodes start

on one machine, but listen on different local port numbers, we will make 5 pre- gRPC request calls from client side before formally recording the data for each set of tests to avoid cold start effects. The following section presents the result of the experiment.

3.2.2 Experimental Result

Number of Nodes	Shortest Time(ms)	Longest Time(ms)	Average Time(ms)
4	4.18	4.92	4.47
7	5.52	6.44	6.02
10	6.35	7.60	7.10

Table 3.1: Time to reach consensus

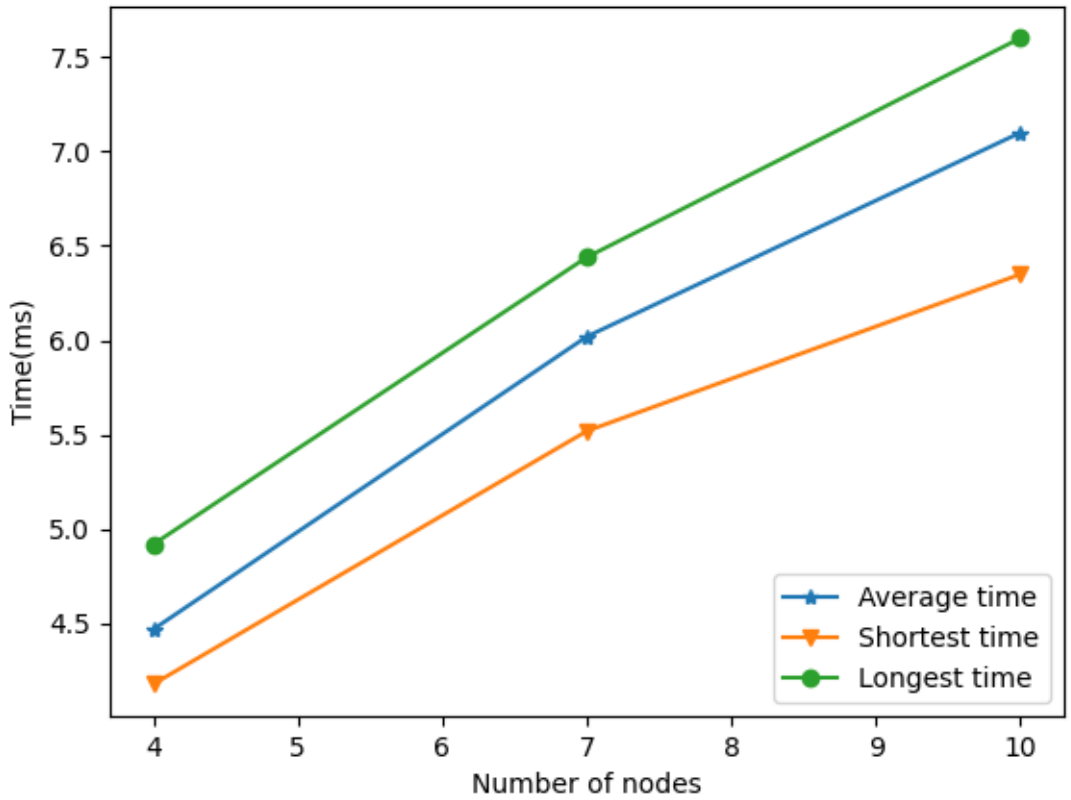

Figure 3.1: Time to reach consensus

In each phase of BFT Baxos, only the Proposer broadcasts to all Acceptors. While the Acceptors respond to the Proposer once with a partial signature to certify the vote or without partial signature to certify the contention. As same as the Hotstuff, the authentication complexity of

BFT Baxos is $O(n)$. Compared with PBFT, since BFT Baxos does not have broadcast phases, the communication complexity is also $O(n)$. The experimental results also demonstrate that the time to reach consensus is linearly related to the number of nodes in the cluster. Therefore, BFT Baxos has good scalability.

Chapter 4

Future Work

4.1 Multi-instance BFT Baxos

In this project, we only implement the single instance version of BFT Baxos. Although it is straightforward to derive a multi-instance version but to achieve the BFT, we introduce many other local variables (retry table, next ballotQC...), and how to coordinate these parameters to work appropriately within different instances might bring some challenges. After the extension to the multi-instance version, we also want to do more performance evaluation, including Workload and DDoS Performance.

4.2 Different random seed strategies in VRF

We now use the Byzantine tolerance timestamp as the random seed for refreshing the VRF and the Byzantine timestamp to implement a random backoff check mechanism. Building Byzantine timestamp requires the Acceptors to tag a timestamp in the Promise message, Preaccept message and Accept message. On the other side, each Proposer retry must carry a large set of timestamps. In the case of intense conflict, this will take up much bandwidth. We can use some new strategy as a random seed for refreshing the VRF, while applying the batch processing idea to the random backoff check mechanism. To generate multiple k and calculate the corresponding multiple backoff times at once timestamp exchange for the use of later backoff to reduce the number of timestamps that need to be carried in the messages.

4.3 Clock synchronization

We know there is no perfect clock, and the nodes' clocks will become increasingly out of sync as the system runs. Since our BFT Baxos relies heavily on a random backoff mechanism to ensure the termination property, we can periodically use Byzantine timestamps to synchronize the clocks.

Bibliography

[1] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[2] Miguel Castro, Barbara Liskov, et al. "Practical byzantine fault tolerance". In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

[3] Pasindu Tennage, Cristina Basescu, Eleftherios Kokoris Kogias, Ewa Syta, Philipp Jovanovic, and Bryan Ford. *Baxos: Backing off for Robust and Efficient Consensus*. 2022. DOI: 10.48550/ARXIV.2204.10934. URL: https://arxiv.org/abs/2204.10934.

[4] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. "Hotstuff: Bft consensus with linearity and responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356.

[5] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. "Byzantine ordered consensus without Byzantine oligarchy". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 633–649.