Byzantine Consensus and Application

Naili(Kevin)

ABSTRACT

The consensus algorithm is to reach an agreement among multiple parties. It ranges from Paxos [6], and Raft [5] to Byzantine consensus, Proof of Work (POW), Proof of Stake (POS), Delegate Proof of Stake(DPoS), and Ripple. Their benefits of decentralization, consistency, and security have driven the development of blockchain networks. In this work, we will investigate one typical Byzantine consensus protocol - Practical Byzantine Fault Tolerance (PBFT) [1] and one application, Algorand [4], which relies on Byzantine consensus protocol. Specifically, we will discuss their assumptions, basic procedures, limitations, and essential properties such as liveness and safety.

1 INTRODUCTION

With the increasing malicious attacks or software errors in model distributed systems, it's essential to equip any distributed system-based software, such as online information services, with a Byzantine-fault-tolerant ability, such that it can function correctly even if some participants in the system are Byzantine faults. This is called Byzantine consensus. Byzantine consensus is a fundamental problem in the distributed system. Formally, it refers to reaching an agreement on a value among n parties, in which up to f can be Byzantine faults - the node can have arbitrary behavior and can be malicious. There are two variants of Byzantine consensus: Byzantine Broadcast (BB) [3] and Byzantine Agreement (BA) [2]. The BB assumes a single broadcaster trying to broadcast a value and targets reaching an agreement on it. While BA assumes each party holds a value and expects all parties will output it eventually. Many protocols have investigated them under different combinations between communication patterns (synchrony, asynchrony, and partial synchrony) and cryptographic assumptions (using digital signatures). And different combinations influence the number of faulty parties the protocol can tolerate. Specifically, BA requires f < n/3 if using a digital signature in partial synchrony or asynchrony network settings. While it only can tolerate f < n/2 parties under synchrony network communication pattern.

PBFT [1] further extends the protocol into a more general and practical case where all the parties run in asynchrony systems to prevent the denial-of-service attack, and the honest client can send many operations to one of a group of parties. The group of parties provides both Byzantine consensus and linearizability. However, it incurs high agreement latency due to many rounds of communications, and it requires fix set of servers determined ahead to prevent Sybil's attacks. In contrast, BA and BB don't rely on a fixed set of servers.

To reduce the agreement latency and allows scaling to many users dynamically without risking Sybil attacks, Algorand [4] is then proposed. Algorand can confirm transactions in a minute and can be safely scaled into many users (500K) using Byzantine Agreement Protocol and Verifiable Random Functions techniques. In the following Section, I will introduce the PBFT and Algorand at Section 2 and 3 respectively. And the summary and conclusion will be presented in Section 4.

2 PBFT

In this section, we introduce the assumptions and basic protocol of the PBFT. We then present some Byzantine faulty behaviors and discuss how to provide safety and liveness under those situations. The overall protocol and corresponding optimization are then given based on our discussion. Finally, we summarize the safety, liveness, and limitations in the last Section.

2.1 Why need PBFT ?

The traditional consensus algorithm such as Raft and PMMC cannot provide Byzantine consensus because the majority $f + 1$ nodes out of the total $2f + 1$ nodes may include Byzantine fault nodes, which could incur the error. For example, they can store different operations under the same slot and arbitrarily reply to client requests. Thus it breaks the system's safety. BA and BB protocols are proposed to tolerate the Byzantine faults nodes, but they are not practical since they only demonstrate theoretical feasibility and are inefficiently used in practice. PBFT is then proposed to provide the Byzantine consensus.

2.2 Assumption and System Model

Many protocols rely on a synchrony network where the communication delay has a bound, which is valuable for the denial-of-service attack where the faulty node is delaying the honest nodes. To make it more practical, PBFT assumes all parties run in an asynchronous system where a network connects all nodes. The network may fail to deliver messages, delay, duplicate, and deliver them out of order. Besides, it assumes independent node failure where the faulty node cannot cause other nodes to be faulty.

Thirdly, it uses cryptographic techniques and assumes public-key cryptography where all parties can sign with a private key and send it to the target server, which could then verify it with the public key. Moreover, it assumes the

faulty nodes are computationally bound and cannot subvert the cryptographic methods.

2.3 System properties

With the above assumption, the system is then designed to satisfy both safety and liveness under the condition where the number of fault nodes $f <= (n-1)/3$, n is the total number of nodes in the system. Safety refers to replicated services satisfying linearizability (behaves like a centralized implementation that executes operations atomically, one at a time). And liveness is defined as the property in which the system can progress under various situations.

The condition can be intuitively explained as follows. Assuming there are total n nodes in the system, f among them are not responding but are not faulty, and the f faulty nodes may be on the $n-f$ side. As a result, the honest nodes $n-f-f$ must outnumber the fault nodes f. So we could formulate the condition as $n-2f > f$, then we get $n > 3f$ and:

$$n = 3f + 1$$

Based on such conditions, any $2f+1$ nodes must intersect at least one non-faulty node. And $2f+1$ is called Quorums.

2.4 Byzantine faulty behavior

We discuss some Byzantine faulty behaviors and how to prevent them from attacking the system. The protocol works under a fixed set of n nodes where f nodes are faulty. And the protocol follows the simple client-server communication pattern, where the client can send operations to one server (primary) and wait for their reply. The servers then provide linearizability properties. There are various worst cases where the fault node can be at the client, primary, and replicas.

2.4.1 Client Faulty. PBFT provides access control to all clients to reduce the faulty client's damage. It denies access to any client if it has sent unauthenticated operations.

2.4.2 Primary Faulty. The primary can be faulty and behave maliciously in multiple ways: (1) assigning different commands to the same sequence number. (2) assigning the wrong commands to the replica. (3) sending the wrong result to the client. (4) ignoring the client's request directly.

As for the first case, the replica could exchange information from the primary with each other and determine whether the primary is honest. It requires one round of communication before sending a prepared message to the primary in PBFT. The protocol run in views. For each view, one replica is the primary, and the others are backups.

In the second case, since the message has the client's signature and each replica can verify the message is from the client by verifying it using the public key.

In the third case, the client cannot trust a single reply from any server. Instead, it requests many replies from multiple

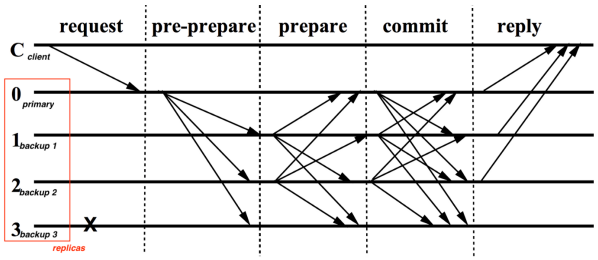

Figure 1: Normal Case Operation

servers so that the client can determine the right reply. In PBFT, it requires at least $f+1$ matching replies. The reason is that $f+1$ nodes must include at least one honest node if they can only tell the truth. If the replies of $f+1$ are matching, then the result is exactly the reply of the honest node.

As for the last case, all replicas should replace the faulty primary and elect a new primary such that the system can still make progress. In PBFT, this is called view change. Each replica has a timer and starts a view change procedure once it is timing out.

2.4.3 Replica Faulty. Replica can be faulty and lie about the commands it received. In PBFT, each replica cannot make progress unless it receives $2f+1$ matching requests from others. Since there are at most f fault nodes and $n = 3f+1$, then f fault replica cannot prevent the system from making progress.

Moreover, the replica and primary can impersonate each other. This is prevented with the cryptography signing.

Gathering all those information, we then present the full PBFT protocol, which can prevent any above attacks.

2.5 PBFT Overview

In this Section, we present the overall protocol. We don't focus much on each phase's communication message details but mainly discuss why the phase is necessary and how it guarantees safety and liveness. The message format and contents can be referred from the original paper.

2.5.1 Normal Operations. We begin by introducing the normal case operation as shown in figure 1, where there are three phases, namely the pre-prepare phase, prepare phase, and commit phase.

Client. The client is assumed to wait for one request to complete before sending the next one. It maintains a monotonically increasing timestamp t to ensure exactly-once semantics and a local timer for retrying. A client sends a request to the primary and waits for a reply. Once it does not receive replies from replicas, it retries by broadcasting the request to all replicas. In exactly-once semantics, all replicas maintain all processed requests, and they will either resend the existing result or execute it as a new request.

Finally, the client waits for $f + 1$ matching replies to get the result. $f + 1$ must contain one honest node since there are at most f fault nodes.

Phase 1: Pre-prepared. After receiving a request message m from the client, the primary firstly assign an order to it, e.g., assign a slot number, then sign and broadcast the pre-prepare message $< (v, nD(m)), m >$ to all other replicas, in which n is slot number, v is view number, $D(m)$ is hash of the message. It only contains the $D(m)$ instead of m in signing to keep the pre-prepare request small and speed up signing. Other replicas can use the view number to determine whether they accept it and store the m locally.

In summary, the Pre-prepare phase is to assign an order to an operation and also sync the messages sent by the client.

Phase 2: Prepare. When a replica receives the pre-prepare message $< n, v, D(m) >$, it accepts the message if and only if

- The signature on m is from the client.
- The hashed value $D(m)$ is truly based on m.
- v is the same as the local view number.
- there is no accepted pre-prepare message at slot n.

After receiving the request, the replica signs and then broadcasts a Prepare message $< v, nD(m) >$ to all other replicas. This is the cross-communication phase where replicas try to determine if the primary is faulty in case of broadcasting different messages to the different replicas. If the replica receives $2f$ Prepare messages matching the one Pre-prepare message sent by primary, this replica can produce a Prepare Certificate on message m at the slot n when the view is v. $(< D(m), v, n >)$

Prepare Certificate indicates that quorums can accept no other $D(m')$ at v, n. The reason why $2f + 1$ guarantees that is that every two $2f + 1$ quorums must interact with at least one honest server, and the honest server never lies. So even if the falulty nodes accept a different message, they cannot form the quorum.

In summary, the Prepare phase is to make sure there is only one message in a slot at a view that can be accepted by quorum.

Phase 3: Commit. However, even if a server produced a Prepare Certificate for $< D(m), v, n >$, it still cannot execute the command and commit it immediately because it doesn't know if others have the Prepare Certificate at $< D(m), n, v >$, and if there is no quorum all having the Prepare Certificate at $< D(m), n, v >$, then executing it may break the safety.

Assume $2f$ replicas execute it after producing a Prepare Certificate and then reply to the client, the f replicas out of which crash, and another f nodes are Byzantine fault nodes. Now the client receives $f + 1$ matching replies and thinks the request is successfully committed. Then the client can issue a read request to the alive $2f + 1$ replicas. However, only f fault nodes have such value, and they may not get a response, while other $f + 1$ honest nodes don't have the value. Thus it breaks the safety of the consensus.

So there is another round of communication to ensure the replica also knows quorum has it.

After a replica produces a Prepare Cerfitifates, it signs and broadcasts a commit message $< v, n, D(m) >$. For each replica, if it receives a $2f + 1$ matching commit message, it knows every quorum must have at least one non-faulty node with Prepare Certificate at $< v, n, D(m) >$. Since the Prepare Certificate guarantees no different messages at the same view and slot number, the message $< v, n, D(m) >$ is then stable and can be executed (if all proceeding operations are already executed) and replied to the client.

In summary, the Commit phase ensures that a quorum of nodes has accepted the deterministic message at a slot number and one view. And even if f nodes crash and f nodes are Byzantine fault nodes, at least one honest node has the deterministic message.

2.5.2 Normal Operation Conclusion. We summarize the key information here:

- Pre-Prepare phase ensures the order.
- Prepare phase further ensures that each message at a slot number and view number is deterministic at any replica with the Prepare certificate. In other words, if decided to commit later, it is the one to be committed.
- The Commit phase is to ensure at least one honest alive node knows which message to execute and commit.

2.5.3 View Change. As presented in Section 2.4.2, once the primary is not responding and acts as a faulty node, other replicas will trigger a view change protocol to move the view from v to $v + 1$ and with a new primary.

The main idea is that they gather information from $2f + 1$ nodes. If a message is committed, the Prepare Certificate is present in at least one of the $2f + 1$ nodes.

The new primary firstly gather message $< v + 1, Pi >$ from all replicas, where P are set of Prepare certificates in node i. Since the primary could be faulty, it will broadcast a new message $< v+1, V, O >$ to all other replicas. V is the set of all view change messages received from replicas, and O is all Pre-prepare messages of all collected Prepare certificate messages. The replica then uses the V to determine if the primary is faulty and re-commits the message which is not committed in the previous view by sending Prepare request for all messages in O.

2.5.4 Garbage Collection and State Transfer. If the service runs for a long time, it will have many stale logs to be collected. They need a new protocol to compact them and save space usage.

So severs periodically decide to take a checkpoint, each server hashes the state of its state machine and broadcasts $< n, D(s) >$ where n is the sequence number of the last executed command and $D(s)$ is a hash of the state. Once a server has $2f + 1$ matching checkpoint messages, it can compact its log and discard old protocol messages. These messages serve as a Checkpoint Certificate, proving the validity of the state.

As for the state transfer, each replica cannot directly accept a state transferred from another node. It needs proof.

2.5.5 Optimization . The PBFT made many optimizations to improve the algorithm's performance during normal-case operations while preserving the liveness and safety.

To reduce the communication overhead, it uses three techniques.

- To avoid sending larger replies to a client. The client request designates a replica to send the result instead of waiting for $f + 1$ replies; all other replicas send replies containing just the digest of the result. With this, the client can verify the correctness of the result.
- To reduce the latency, the replica can execute it and send a tentative reply message to the client; once the client receives $2f + 1$ matching tentative replies, the message can be guaranteed committed in the third phase. If the executed request is aborted due to a view change, the replica will revert the state to the original.
- For the read-only transactions, the client broadcasts it to all replicas; each replica verifies the client's signature of the message and executes it immediately; the client then waits for $2f + 1$ matching replies. If the client cannot collect so many replies due to other write congestion, the client resends the read request as a read-write request.

Moreover, PBFT also uses message authentication codes (MACs) to authenticate the message in normal operation to reduce the latency because MACs can be computed three orders of magnitude faster than digital signatures. While digital signatures are only used for view change and new view messages.

2.6 Safety

In summary, the safety of PBFT relies on the fact that

- Faulty replica f < n/3
- Replica runs deterministically and must start in the same state.

2.7 Liveness

The liveness relies on synchrony with bounded delay, where the delay is defined as the time between the moment when a message is sent for the first time and the moment when its destination receives it. Besides, view change protocol prevents primary from failure, such that the system can make progress.

2.8 Limitations

Although PFBT can tolerant any failures ranging from benign to malicious as long as it only affects less than 1/3 of replicas, it has some limitations:

- All participant nodes are fixed, and adding new nodes may suffer from Sybil attacks.
- Committing a single transaction requires high latency due to $O(n^2)$ rounds of communications.
- Cryptography operations are slow and further slow down the transaction speeds.

3 ALGORAND

Algorand is based on Byzantine Agreement protocol while integrated with Verifiable Random Functions to achieve low latency and high scalability.

3.1 Related techniques

3.2 Assumptions

3.3 Protocol Overall

3.4 Safety and Liveness

3.5 Performance and Limitations

4 CONCLUSION

We are trying to investigate the important properties of Byzantine consensus protocol - PBFT and one application that replies to Byzantine consensus protocol.

REFERENCES

[1] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, Margo I. Seltzer and Paul J. Leach (Eds.). USENIX Association, 173–186. https://dl.acm.org/citation.cfm?id=296824

[2] Danny Dolev and H. Raymond Strong. 1983. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.* 12, 4 (1983), 656–666. https://doi.org/10.1137/0212045

[3] Vadim Drabkin, Roy Friedman, and Marc Segal. 2005. Efficient Byzantine Broadcast in Wireless Ad-Hoc Networks. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*. IEEE Computer Society, 160–169. https://doi.org/10.1109/DSN.2005.42

[4] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. *IACR Cryptol. ePrint Arch.* (2017), 454. http://eprint.iacr.org/2017/454

[5] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[6] Robbert van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3 (2015), 42:1–42:36. https://doi.org/10.1145/2673577