Tow-Phase Commit Rule for Blockchain Consensus

Taining Cheng, Shilei Zhang, Jinhong Zhang, and Jing He(✉)

School of Software, Yunnan University, Kunming, China
{tncheng,zsl_2019,zjhnova}@mail.ynu.edu.cn, hejing@ynu.edu.cn

Abstract. All Byzantine agreement protocol is widely used in the permissioned blockchain as the core of consensus. At present, there are already many research works on the agreement problem, such as the state-of-the-art HotStuff solves the byzantine fault-tolerant problem with linear complexity. Still, it takes more message round to commit a transaction. Therefore, we transform the agreement problem as a broadcast problem and solve it with less round theoretically, also proving its safety and liveness. Moreover, the experiments show that our protocol has less latency under the same throughput performance as HotStuff.

Keywords: Byzantine fault-tolerance · Consensus · Blockchain

1 Introduction

The emergence of blockchain and many decentralized applications has led researchers to focus on secure and efficient Byzantine fault-tolerant protocols for large-scale networks [9]. As a distributed ledger, blockchain maintains the consistency of the ledger by state machine replica (the replicas from the same state and receives identical input sequences to reach the same state), which was proposed by Lampert [9]. Researchers have done a lot of work from theory to practice. An unpredictable network condition and arbitrary behavior of malicious makes reaching agreement more difficult; malicious can arbitrarily delay message arrival time to interface with the entire system to reach consensus. Therefore, the agreement protocol needs to ensure both safety and liveness under any network condition [5], but the classical FLP "impossible" theory has proved that deterministic agreement protocol does not exist under asynchronous network [6]. The researcher introduces the timing assumption to constrain the network further to obtain a deterministic protocol [13], as much state-of-the-art synchronous protocol raft [12], which assumes the message delay within a known upper bound Δ. Conversely, if the message delay can be arbitrarily controlled by malicious at most unknown bound, which is called asynchrony. Both above assumptions are too extreme; generally, one typical compromise is partial synchrony that assume the adversary can fully the network to be asynchrony before the event called GST (Global stabilization time) [13], after which the network becomes synchrony. Although the partially synchronized PBFT protocol has proven to have a good performance after GST [4], it lacks liveness before GST. This means that when the network is unstable and rarely synchronized, the protocol stagnates and does not provide services. Worse more,

Z. Zeng et al. (Eds.): ECIT 2022, AHE 11, pp. 531–541, 2023.
https://doi.org/10.2991/978-94-6463-005-3_53

the fragile liveness brought by timing assumptions remains theoretical (too small timeout leads to trivial view-change, too large leads to lower TPS) [16]. Timing assumption directly makes the agreement protocol lose the responsiveness (namely, the performance depends on the real network latency) [1, 9].

In contrast, the random asynchronous protocol achieves strong robustness through higher communication costs [2, 3]. This protocol does not depend on any timing assumptions, but can ensure liveness and responsiveness during running; such as HotStuff and its variant requires only seven or more round message exchanges to reach agreement [17]. Consequently, asynchronous BFT tends to bypass the FLP "impossibility" by randomness, costing more communication rounds. Well, the study if asynchronous consensus remained theoretical for a long time until HoneyBadgerBFT and Dumboo was proposed by Miller [11], Bingyong, which provides practical usability and relatively low complexity [7]. However, there is still a big gap between the best synchronous protocol. Therefore, a combination of deterministic and random approaches is a natural way to improve asynchronous protocol in optimistic times and robustness in pessimistic times [10]. The previous work optimistic method in asynchronous network was given by Kursawe attempted to switch the synchronous consensus to asynchronous by fall-back [8]. This approach is called view-change in PBFT, but with a difference, PBFT depends on timing assumptions. The intuition behind this is that network can become unstable or attacked in the real world, but there is enough time for the network to keep in sync, message delay within Δ, the optimistic path can be used under stable.

Whether it is deterministic or random scheme, the combination, the former in an indispensable path for transaction commit, dramatically improves the overall performance if the deterministic path can be optimized, because the efficiency is dominated by it in most cases. In this paper, we make a small step on the road to deterministic protocol. First, we formalize the consensus problem as a broadcast problem [8], namely, a designated replica as proposer atomic broadcasts the transactions to all, and completes the commit after completing two-phase (prepare, commit described below) consensus with linear complexity under threshold signature scheme setting. The consistency under same view and across-view are guaranteed by the prepare and commit phase respectively [15]. Furthermore, protocol availability (liveness in agreement problem) is ensured via "leader change".

The reset of this work is organized as follows: the preliminary protocol and problem definition are presented in Sect. 2. The detail of phases and proof of protocol is illustrated in Sect. 3, and more experiments are given in Sect. 4.

2 Model

2.1 Notations and Data Structure

Following the solutions [4, 17], as a permissioned blockchain platform that consists of n replicas, f of which are adversary and $f < n/3$, otherwise honest, the adversary may

corrupt replicas are Byzantine faults and deviate from the protocol; and they control the message delivery times, but the message among honest replicas is eventually delivered. In addition, each replica has a public key certified by a public-key infrastructure (PKI) [15], the replicas have all-to-all reliable and authenticated communication channels.

2.1.1 Cryptographic Assumptions

We assume each replica has a public key provided by a trusted dealer; we do not consider the security problem of cryptographic scheme in order to focus on the distributed aspect of the problem. In this paper, we use threshold signature scheme, where a set of signature shares for message from t (the threshold) distinct replicas among n parties can be combined into one threshold signature of same length of common signature scheme, we denote a threshold signature share of a message m signed by a replica i as $< m >_i$. For our protocol, once any of n replica has threshold signature, it means the owner of threshold signature has seen enough votes for the proposal.

2.1.2 Collision-Resistance Hash

We assume a hash function $H(\cdot)$ that can map an input of arbitrary size to an output of fixed size. Specifically, $H(\cdot)$ is Collision-resistance, no probabilistic polynomial algorithm adversary can generate a couple of distinct inputs x_1 and x_2 subject to $H(x_1) = H(x_2)$.

2.1.3 View Number

View is the unit of protocol runs, specifically each replica commits at most once and keeps store and update the current view number as v, which is initially set to 0.

2.1.4 Block Structure

The Block is represented by a tuple $B = \left[id, v, txs, proof\right]$, where $id = H(block)$ is the digest of current block and identification, v is the view number of block, txs is a batch of uncommitted transactions, $proof$ is defined below.

2.1.5 Proof

A *proof* is evidence that there are enough replicas to receive the message from the proposer; it is formed by a quorum of $n-f$ "vote" message. According to the type of "vote" message, There are two types of *proof* . First, The *proof* of block B is threshold signature

proof generated by the proposer, and denote by a tuple $proof = [type, id, v, ssig]$, where id, v, r is the same as block's and $ssig$ is threshold signature produced by combining the distinct replica's signature shares on the block identification($\langle id \rangle$), and $type = vote$, we say a block is certified (it can be committed) if the $proof$ is existed. Second, timeout or blame $proof$ of view v, denote by tuple $proof = [type, states, v, ssig]$, containing a threshold signature $ssig$ on view v and $type = blame$. Same as above but slightly different, the signature can be generated once the proposer receives a quorum $n - f$ blame message, which is the threshold signature share($\langle v \rangle$) sent by replica, it also contains a state set $states = \{proof_n, 0 < n \leq n - f\}$ of $n - f$ block certification, which is used to synchronize the block cross-view, the element of set $states$ is the most. Finally, we will use $x.y$ to denote the element y of x below.The Block is represented by a tuple $B = [id, v, txs, proof]$, where $id = H(block)$ is the digest of current block and also identification, v is the view number of block, txs is a batch of uncommitted transactions, $proof$ is defined below.

2.2 Problem Formulation

Broadcast Problem. Here, we assume a designated replica, often called the leader that has some input block b, A protocol that solves the Broadcast problem must have the following properties [1, 14].

- (Agreement): No two honest replicas commit different blocks.
- (validity): If the leader is honest, then b must be the committed block.
- (termination): All honest replica must eventually commit a block terminate.

In this paper, we formalize the consensus algorithm as a **Broadcast Problem**. All replicas vote on the proposal of the designated leader (proposer) view by view.

3 Protocol

In this section, we elaborate on the detail of the protocol below. To satisfy the property of **Broadcast Problem**, the protocol can be separated into two parts, The first sub-algorithm is running in a steady state, another sub-algorithm is used to ensure quorum replicas "see" and commit the block when the network becomes asynchronous, or the leader is malicious. This pattern has linear communication complexity equipped with threshold signature scheme; it is like Hotstuff and originates from the PBFT.

Algorithm 1. steady state for replica r_i

1: Let *proposer* is the leader of current view v_c
2: **Initial:** $v_c \leftarrow 0, proof_p \leftarrow \perp, \tau \leftarrow rand$
3: upon $runView(v, proposer)$
4: if $proposer = r_i$ then
5: wait $|newView| = n - f$ or $|Blame| = n - f$
6: if $|newView| = n - f$ then
7: $proof_m \leftarrow (max(proofSet), type = vote)$
8: if $|Blame| = n - f$ then
9: $proof_m \leftarrow (max(proofSet), type = vote)$
10: send $block = [id, v, txs, proof_m]$ to all

11: wait until $|Prepare_s| = n - f$
12: $proof \leftarrow$ combining $Prepare_s$
13: update local $proof_p \leftarrow proof$
14: send prepare $proof$ to all replicas

15: wait until $|Commit_s| = n - f$
16: $proof \leftarrow$ combining $Commit_s$
17: update local $proof_c \leftarrow proof$
18: send prepare $proof$ to all replicas
19: commit $block$
20: update view $v_c = v_c + 1$

21: upon $Clock()$ do
22: if τ is timeout
23: $Blame \leftarrow (proof_p, type = blame)$
24: send $Blame$ to all replicas

3.1 Phase

To better understand the algorithm, Algorithms 1 and 2 presents the normal procedure that makes progress when the leader is honest. Before the replica moves forward to the next view, all replicas will send its local $proof_p$ in *newView* message. Specifically, the proposer of each view can be elected following the order of replica id (it can be any more safety scheme). Then the proposer proposes block B that extends a block certified by the highest *proof* with the highest view number among they received $n - f$ $proof_p$ from other replicas. When a replica gets a proposal B, it first verifies the validity of cryptography; for the agreement property, the view of *proof* in block is matched for the local variable $proof_p$ if the $proof.type = vote$. For termination property, the replica also needs to compute the highest *proof* among *states* set to matched the *proof* in block B when $proof.type = blame$. In brief, before voting (sending prepare signature share to proposer), it verifies that at least one of the following two conditions is satisfied:

- $B.v = B.proof.v + 1$

- $B.v = B.proof.v + 1$ and

  $B.proof.v \geq \max\{proof_m.v | proof_m \in proof.states\}$ if $proof.type = Blame$.

Algorithm 2. message handler for replica r_i

```
upon receiving block from proposer do
    if proof_c =⊥ ⋁ v_c = proof_c.v + 1 then
        if proof =⊥ ⋁ block is valid then
            reset τ
            Prepare_s ←< block >_i
            send Prepare_s to proposer

upon receiving prepare proof do
    if prepare proof is valid then
    reset τ
    update local proof_p ← proof
    Commit_s ←< block >_i
    send Commit_s to proposer

upon receiving commit proof do
    if commit proof is valid then
        reset τ
        update local proof_c ← proof
        commit block
        update view v_c = v_c + 1
        send newView to next proposer

upon receiving Blame do
    wait until |Blame| = f + 1
        Blame ← (proof_p, type = blame)
        send Blame to all replicas
```

In other words, either proposal B contains a *proof* of the proposal in previous view or a *proof* of previous view with a set of at least $n-f$ *proof*, the replica can use the set to verify the highest commit view of the whole system. After satisfying agreement and termination, replica replies a signature share on the block to proposer, and receives the *Prepare proof* once proposer collects the share on block from $n-f$ replicas, *Commit proof* is the same as *Prepare proof*. Finally, a replica can commit the transactions in block B when it sees *Commit proof*, and advancing the view.

3.2 Proof of Protocol

3.2.1 Agreement Proof

Lemma 1. For any two *proof* within a view v, such as $proof_1, proof_2$, we have $proof_1.v = proof_2.v$.

Proof: We proved by a contradiction, assume exist $proof_1.v \neq proof_2.v$. Because at least $n-f = 2f+1$ share (threshold signature share) are required to form a *proof*. The

Table 1. Complexity of protocol.

	Normal	View Change	Latency
HotStuff	$O(n)$	$O(n)$	7 round
PBFT	$O\left(n^2\right)$	$O\left(n^2\right)$	3 round
Ours	$O(n)$	$O\left(n^2\right)$	5 round

equation $2f + 1 + 2f + 1 - (3f + 1) = f + 1$ must be true if exist two *proof*. We can see that there must be an honest replica who send share for two different proposals, it's violating the assumption that Byzantine problem assume there only f adversarial.

3.2.2 Termination Proof

Lemma 2. At any view v, if the proposer is honest, a bounded time T must exist such that a proposal is committed during the time interval T after GST. Otherwise, When the replica enters the view with a malicious proposer, the replica will commit nothing until its timeout to enter a view v' with an honest proposer.

Proof: According the Algorithm 2, a replica entering a new view v by send a *newView* view message with $proof_p$ in view $v - 1$, the proposer collects $(n - f)$ *newView* message and finds the highest $proof_m$ as a proof in block to persuade other replicas in v. Thus, all honest replicas will send share back to the proposer when a replica's local $proof_p$ can match *proof* in new block, because *proof* is formed by at least $f + 1$ honest replicas. After the proposer collects $n - f$ shares and generates *Prepare proof* for proposal, all replicas will see *proof* and vote in the following phase and continue to advance the view within time T. As for before GST, all replicas will trigger "leader change" once an honest send *Blame* message to others because of timeout or suspicion of current leader, the new proposer will follow algorithm 1 to combine $n - f$ *Blame* to form *proof* of blame, and then telling every other where to commit in view v' if new proposer is honest after GST. Finally, all honest replicas are synchronized in the global view and committed.

3.3 Complexity Analysis

It can be seen from the phase of protocol that the communication complexity is linear when normal commit is in progress (the proposer is honest and after GST), because the pattern of communication is all to one or one to all and the message complexity within 1 round is constant. Furthermore, we use threshold signature to decompose one all to all message exchange round into two rounds, and every message contains n *proof* to satisfy safety. Finally, we get 5 round latency and quadratic "view change" complexity. The comparison of PBFT and HotStuff with ours is shown in Table 1.

4 Copyright Form

In this section, we evaluate the throughput and latency of our protocol, which is implemented by Golang; we deploy it to different numbers of instances to measure the scalability and robustness. In addition, we conduct faulty attack on the protocol and measure the performance and latency. Specifically, our latency measurement is end-to-end that the time elapsed from a transaction sending until safe commit, shown as Eq. (1):

$$latency = T_c - T_p = T_{btx} + T_{ctx} + T_{bb} \tag{1}$$

where T_c, T_p respectively denote time of commit and proposal, it can be further decomposed into three stages: transaction broadcast, transaction consensus, block broadcast. The throughput is calculated as Eq. (2):

$$tps = \frac{|txs|_{\Delta t}}{\Delta t} \tag{2}$$

where $|txs|$ is total number of transactions committed during time interval Δt.

4.1 Best Performance

As we can see from Fig. 1, it shows the performance of ours and several classical protocols with varying number of replicas (4, 8, 16, 32, 64), and there is no one replica be malicious.

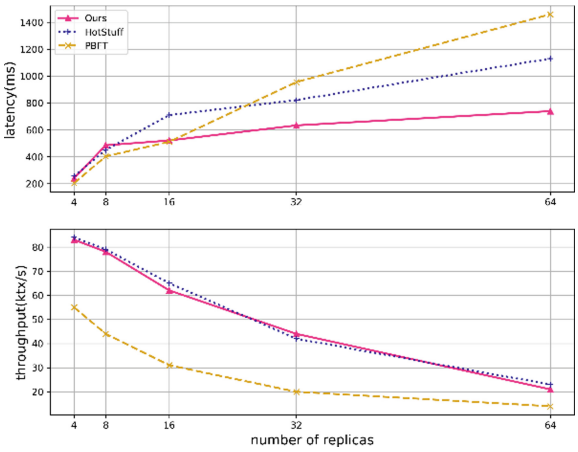

Fig. 1. Scalability of different protocol

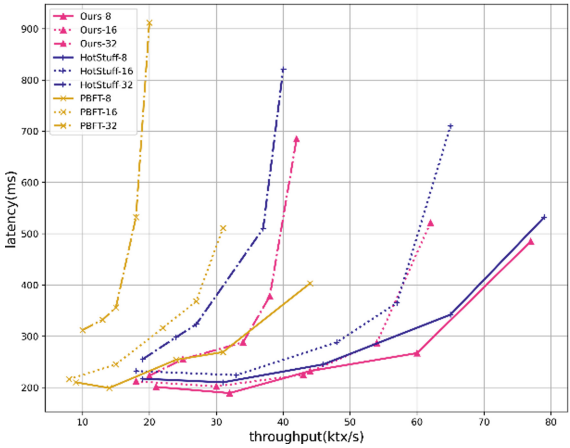

Fig. 2. Throughput-latency performance with different replica size of 8, 16, 32

Obviously, our protocol's latency is gradually better than other two as the number of replicas increases. The PBFT's latency is the best since PBFT requires fewer message round to reach agreement than other two, and the total messages do not reach the bandwidth's upper limit. After the number of replicas increases, protocol (ours) that requires fewer message rounds performs best. At the same time, our throughput is close to the best HotStuff under any network condition, but also definitely better than PBFT.

Figure 2 illustrate the throughput-latency performance under different replica size. The overall performance of Ours and the state-of-the-art HotStuff is ahead of classical PBFT. Particularly, latency increases with the increase of TPS until it reaches the upper limit, an external component that provides necessary services, such as the bandwidth limit and transaction buffer size, etc.

4.2 Faulty Attack

In this setting, we run these protocols with a fixed total size of replicas and varying faulty replicas. The results of performance are presented in Fig. 3. Clearly, we can conclude that the performance of all protocols has a certain degree of loss, HotStuff has the best performance, ours is a little bit worse, PBFT is the worst, Because the timeout occurs when proposer is adversarial, all replicas performance will be lost due to "view synchronized" phase. Surprisingly, we have the best performance in terms of latency and throughput when a small number of replicas is faulty. The reason is that the honest is proposer most of time; it only needs $n - f$ vote from all replicas, the fault tolerance threshold is much larger than actual adversarial, it doesn't take long to recover.

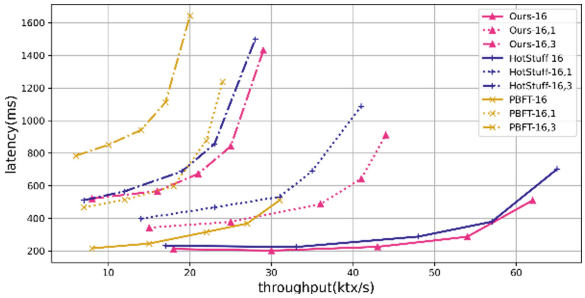

Fig. 3. Throughput-latency performance with replica size of 16 and adversarial size of 1, 3.

5 Conclusions

In this paper, we present an optimized agreement protocol with threshold signature; it takes less time to commit transactions with higher throughput under moderate network conditions. We implement and experimentally measure the performance of our protocol with classical scheme to validate our theoretical analysis. However, there is still some exploration work in the future, such as further reducing the complexity of agreement when the network is asynchronous or proposer is malicious.

Acknowledgements. This work was supported in part by the National Natural Science Foundation of China under Grant 62162067, 62101480, 61762089, 61763048 and in part by the Yunnan Province Science Foundation for Youths under Grant No. 202005AC160007.

References

1. Abraham I, Malkhi D, Spiegelman A (2019, July). Asymptotically optimal validated asynchronous byzantine agreement. In: Proceedings of the 2019 ACM symposium on principles of distributed computing, pp 337–346
2. Bracha G (1987) Asynchronous Byzantine agreement protocols. Inf Comput 75(2):130–143
3. Cachin C, Kursawe K, Petzold F, Shoup V (2001) Secure and efficient asynchronous broadcast protocols. In: Kilian J (eds) Annual international cryptology conference, pp 524–541. Springer, Heidelberg. https://doi.org/10.1007/3-540-44647-8_31
4. Castro M, Liskov, B (1999, February) Practical byzantine fault tolerance. In: OsDI, vol 99, No 1999, pp 173–186
5. Défago X, Schiper A, Urbán P (2004) Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR) 36(4):372–421
6. Fischer MJ, Lynch NA, Paterson MS (1985) Impossibility of distributed consensus with one faulty process. J ACM (JACM) 32(2):374–382
7. Guo B, Lu Z, Tang Q, Xu J, Zhang Z (2020, October) Dumbo: Faster asynchronous bft protocols. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, pp 803–818
8. Kursawe K, Shoup V (2005) Optimistic asynchronous atomic broadcast. In: Caires L, Italiano GF, Monteiro L, Palamidessi C, Yung M (eds) International colloquium on automata, languages, and programming, pp 204–215. Springer, Heidelberg. https://doi.org/10.1007/11523468_17

9. Lamport L, Shostak R, Pease M (2019) The Byzantine generals problem. In: Concurrency: the works of Leslie Lamport, pp 203–226
10. Lu Y, Lu Z, Tang Q (2021) Bolt-dumbo transformer: asynchronous consensus as fast as pipelined BFT. arXiv preprint arXiv:2103.09425
11. Miller A, Xia Y, Croman K, Shi E, Song D (2016) The honey badger of BFT protocols. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 31–42
12. Ongaro D, Ousterhout J (2014). In search of an understandable consensus algorithm. In: 2014 USENIX annual technical conference (Usenix ATC 2014), pp 305–319
13. Pease M, Shostak R, Lamport L (1980) Reaching agreement in the presence of faults. J ACM (JACM) 27(2):228–234
14. Rodrigues L, Raynal, M (2000) Atomic broadcast in asynchronous crash-recovery distributed systems. In: Proceedings 20th IEEE international conference on distributed computing systems, pp 288–295. IEEE
15. Shoup V (2000) Practical threshold signatures. In: Preneel Bart (ed) EUROCRYPT 2000, vol 1807. LNCS. Springer, Heidelberg, pp 207–220. https://doi.org/10.1007/3-540-45539-6_15
16. Spiegelman A (2020). In search for an optimal authenticated byzantine agreement. arXiv preprint arXiv:2002.06993
17. Yin M, Malkhi D, Reiter MK, Gueta GG, Abraham I (2019) Hotstuff: BFT consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM symposium on principles of distributed computing, pp 347–356, July 2019