Chained Tendermint: A Parallel BFT Consensus Mechanism

Lei Lei*, Chunjia Lan*†, Le Lin*†
Neng Lian Tech Ltd., Shanghai, China
†Corresponding Auther
{lei.lei, chunjia.lan, linle}@nenglian.com

***Abstract*—A number of well-known BFT (Byzantine Fault Tolerant) algorithms, such as Tendermint, Casper, HotStuff, and Grandpa, have been developed in recent years to solve the problem of consensus. Of them, Tendermint can ensure instant finality but needs multiple rounds to commit a block. Casper and HotStuff both need a constraint on the direct parent to commit to a block. Grandpa uses ghost function to extract a common block from a vote set, but it is not based on a chained model and does not use signature aggregation to boost performance. We propose a consensus algorithm based on the theories of Tendermint and HotStuff. It is based on the same network model (each message sent out can reach every replica eventually, the adversary can hold it but not forever) and adversary assumption (the adversary can do Byzantine behavior like not proposing block or sending vote, or sending contradictory votes) as in Tendermint, but does not have the feature of instant finality. The result is an increase in throughput, since the blocks are handled in the chained model, the processes of handling each block can overlap.**

***Keywords*—blockchain, BFT consensus, pipeline**

I. Introduction

Unlike the model with faulty process [11] with uncertain process result, blockchain is the model that every replica runs a sequence of transactions that each of them is a deterministic process. As a state transition machine, after a certain sequence of deterministic processes [6][12], each replica should reach the same state. The consensus algorithm is to reach the goal of confirming the sequence of transactions. Each block is a batch of transactions, and the consensus is reached at the unit of the block. The Byzantine fault tolerance (BFT) consensus algorithm is used in a Byzantine environment in which at most f replicas are Byzantine replicas that can disobey the protocol and send out arbitrary message. In this model, the total number of replicas n in environment should be at least $3f+1$. The BFT consensus algorithm defines a set of rules that $n-f$ honest replicas should obey. By using these rules to react to certain circumstances, they eventually reach a consensus on each block of the chain regardless of what the f Byzantine replicas do. They can pretend to be offline when they should be online, or send inconsistent votes with its previous vote, or broadcast two different votes at the same round.

The network model is an asynchronous P2P network, by assumption, there could be some delay of the messages, but no adversary can hold a message forever, which means each message should be eventually received by the recipient.

The feature "instant finality" divide the BFT consensus algorithms into two categories. Instant finality means if the node is at height h, the block of the previous height must have been finalized and immutable. The ones with instant finality: such as Tendermint [1] and HoneyBadger [8]. The ones without instant finality: such as CasperFFG [3], Grandpa [4] and chained HotStuff [2]. Though these protocols lose instant finality, they have better throughout, for they are designed to commit a chain which contains multiple blocks a time rather than commit only one block once a time. Though some business model needs a strictly fixed checkpoint height that the consensus protocols without instant finality may not fit, they fit other models well. These consensus algorithms pursue the two goals below:

1) better performance when the chain does not fork and the network is good;

2) better convergence (which means the honest replicas quickly switch to the same chain) when the chain forks.

Our work is to present a new BFT consensus protocol that is also without instant finality, and it is inspired by Tendermint and HotStuff. We present the protocols in three stages: the first stage is the basic Parallel Tendermint protocol which allows the node enter the next height first and send the previous *Precommits* later, thus the two phases of voting are parallel with each other and also parallel with the broadcast of the blocks.

The second is the optimized Paralell Tendermint protocol described in Section IV. The key thought is to add the *Precommit* into the *Prevote* message to let the voter tell the aggregator his locked block, so that the aggregator aggregates the *Prevotes* to form a PoLC (Proof of Lock: $n-f$ *Prevotes* for the same block), meanwhile he can also aggregates the *Precommits* to form a PoC (Proof of Commit: $n-f$ *Precommits* for the same block) if the *Precommits* are for the same locked block.

The disadvantage of the second protocol is that there are not always $n-f$ voters with the same locked block send their votes to the aggregator at the same time. So the third protocol named Chained Tendermint described in Section V is to let each block contain a LockQC, which is the PoLC of the proposer's local locked block, the voter sees this block and judge if he can accept the proposer's locked block or not. If he can accept, he sends "*Prevote* and *Precommit*" like the second protocol, else he sends "only *Prevote*".

II. Safety and Liveness of Tendermint

Any consensus algorithm needs to satisfy two requirements, safety and liveness, to ensure correctness. Satisfying the demand for safety involves proving that two blocks on different chains cannot both meet the requirement of commit. Liveness is proved by showing that the replica cannot be deadlocked. Under any condition, the participant of the consensus protocol should always have ways to continue and eventually reach consensus with the other replicas.

The condition to commit a proposal block of round r at height h in Tendermint is that there are $n-f$ Precommits of round r for this block, and the proof of safety involves showing that there cannot be two $n-f$ Precommit sets at the same height that are intended for two blocks. This is proved by contradiction in two conditions:

1) Suppose these two sets are in the same round. Then more than f replicas precommit for two blocks at the same round, which violates the premise of BFT consensus protocol.

2) The two sets are not in the same round, Tendermint shows that if the set in the lower round occurs, the set in the higher round cannot occur. To satisfy this, Tendermint achieves a stricter goal:

Theorem 1 (Theorem of Tendermint). If there are $n-f$ *Precommits* of a specific block in any round, $n-f$ *Prevotes* are not available for another block in later rounds.

Theorem of Tendermint can be proved by contradiction as follows: $n-f$ *Precommits* for the same proposal block in the same round r means that more than $n-2f$ honest replicas are locked on this block in round r; if there are many sets of $n-f$ prevotes for other blocks in later rounds, among these sets, we define s as the one with smallest round r^* and $r^*>r$. Because the size of the intersection of set s and $n-f$ *Precommits* of round r is at least $f+1$, there should be at least one honest replica that violates its commitment.

For liveness, we should ensure that the locked blocks in lower rounds always have a chance of being unlocked. In Tendermint, if a replica has received a $n-f$ *Prevotes* for another block or *nil* in a round that is higher than its locked round, it can unlock its locked block and becomes legitimate to prevote other blocks.

In our protocol, cancel the mechanism of multiple rounds at each height and do not require that the block at a height be finalized before beginning the next height, but we want this theorem to still hold:

Theorem 2 (Throrem of Parallel Tendermint). *If there is a proof of commit of some height h for a block b, there is not a proof of lock of a higher* height h′ *for a block* b′ *that is on another chain, where* h′ ≥ h.

III. Basic Protocol

A. Parallel Tendermint Procotol

The main thought of the "Parallel" feature is to flatten the rounds in Tendermint into the heights, and change the requirement to enter the next round from: the replica has to collect $n-f$ *Prevotes* of current round and send out the *Precommit* (If the $n-f$ *Prevotes* are for the same block it precommit this block, else it precommit *nil*), to: the replica is allowed to enter the next round first and collect the PoLC later, it can precommit for previous round if it receives the PoLC of that round and the PoLC is still acceptable. This thought is to make the four phases --broadcast of the proposal block, broadcast of the *Prevotes*, broadcast of *Precommits*, execution of the proposal block – completely parallel, as shown in Fig. 1.

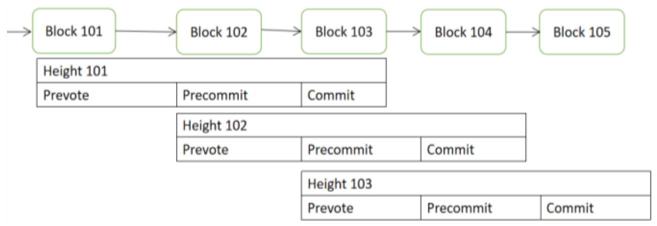


Fig. 1. The normal pace of Parallel Tendermint

Change of the pace affects analysis of safety and liveness.

For safety, since the replica is allowed to enter next height and prevote the next block without having made *Precommit* of current height, we should let it able to send previous *Precommits* when it collects PoLC of any previous height, otherwise no *Precommit* can be made and no block can be committed. One of the principle that the replica should obey is that its later *Prevotes* should be consistent with its previous *Precommit*. Thus it is optimistic to prevote first and precommit previous blocks later. In some situation, the network splits and the chain forks, it is possible that another previous block on the other chain fork that obtains the $n-f$ *Prevotes*, in this case, the replica cannot precommit that block because this *Precommit* is contradictory with its "later" *Prevote*. Though the *Prevote* is made earlier in time base, it is later in height base. It is a Byzantine behavior if the replica sends a *Precommit* for a block of height h , and sends a *Prevote* for another block on the other chain fork of height $(h'>h)$, if there is no PoLC of height that is in the range $(h,h']$ that can let the replica unlock or updates its locked block.

We define an acceptable PoLC is the PoLC that the replica can send *Precommit* for. The logic of judging whether a received PoLC is acceptable is summarized in Table I. An example is also given in § A.

The "Revert" action is more complicated than the "Unlock" action in Tendermint, which simply means the replica unset its locked block and becomes having no locked block. In our protocol, the "Revert" action means the replica reverts its locked block to the last locked point in the common chain of its current prevoting chain and the block of the received PoLC.

The situation of updating locked block but not sending *Precommit* for it can be viewed as an optimization to Tendermint Protocol. In Tendermint protocol, if a replica is in round *r* and the round he locked his current locked block is *r'*, and he receives a PoLC for a block *b* of round *r** and *r'<r*<r*, in this case, he is only allowed to unlock, but not allowed to update his locked block to *b*. The reason is: he cannot send *Precommit* for *b* with either round *r** or round *r*. First we discuss if he sends a *Precommit* for *b* with round *r**. The replica prevoted for his locked block between *r** and *r*, the

Precommit of round r* for b contradicts with these *Prevotes*. Second we discuss he sends the Precommit for b with round r. This breaks the safety of the Protocol. For the *Precommit* for some block b of round r should mean the voter sees the PoLC of exact the same round, not any other previous round. If we change the meaning of *Precommit* to "This is currently the block with the PoLC of the greatest round that I have seen", the $n-f$ *Precommits* of some round cannot deduce to: $n-f$ *Prevotes* of later rounds cannot occur. Because the replicas can change their locked block and prevote other block, thus there could be another block with $n-f$ *Precommits* of some later round, the safety is thus break.

TABLE I. THE HANDLE ON RECRIVING A POLC FOR A BLOCK OF HEIGHT h

Situation 1	**Locked height<Switching height ≤last prevoted height**		**Locked Block**	**Acceptable**
On same chain with Locked Block	h ≤Locked height		Unchanged	N
	Locked height< h <Swtiching height		Update	N
	Switching height ≤ h < last prevoted height	On same chain with last prevoted block	Update	Y
		Not on same chain with last prevotedblock	Update	N
	last prevoted height≤ h		Update	Y
Not on same chain with Locked Block	h ≤Locked height		Unchanged	N
	Locked height< h		Revert	N
Situation 2	**Switching height ≤Locked height≤last prevoted height**		**Locked Block**	**Acceptable**
On same chain with Locked Block	h<Switching height		Unchanged	N
	Switching height≤ h ≤Locked height		Unchanged	Y
	Locked height≤ h <last prevoted height		Update	Y
	h ≥ last prevoted height		Update	Y
Not on same chain with Locked Block	h ≤Locked height		Unchanged	N
	Locked height< h		Revert	N

What if the replica does not send *Precommit* for b, but just set it as its locked block quietly? We analyze this change from safety and liveness. As for safety, since the replica does not send out previous *Precommit*, there is no *Precommit* contradicts with its *Prevotes* in "later" rounds. What about it changes its prevoting block? In Tendermint protocol, the change can be explained as: since I have seen a PoLC for another block with greater round, I can deduce $n-f$ *Precommits* for my locked block cannot occur, thus I can unlock it and prevote another block safely. This applies to the prevoting to new locked block too. As for liveness, since the rule changes to: the replica can only prevote the updated locked block rather than prevote any block, does this cause liveness problem? Suppose the new PoLC is the one with greatest round in the whole network, it will eventually reach all replicas, so all replicas become locking to it, they all prevote it in later rounds, it will be committed. Suppose there is another PoLC with greater round in the network, and PoLC will eventually let all replicas lock to it.

From analysis above, we can see the optimism of sending *Prevotes* of "future heights" has the downside: it causes the replica unable to make *Precommit* for previous block on another chain even if the replica receives the PoLC of it, thus possibility of commits of previous blocks of other chain forks decreases, though the replica can update its locked block quietly.

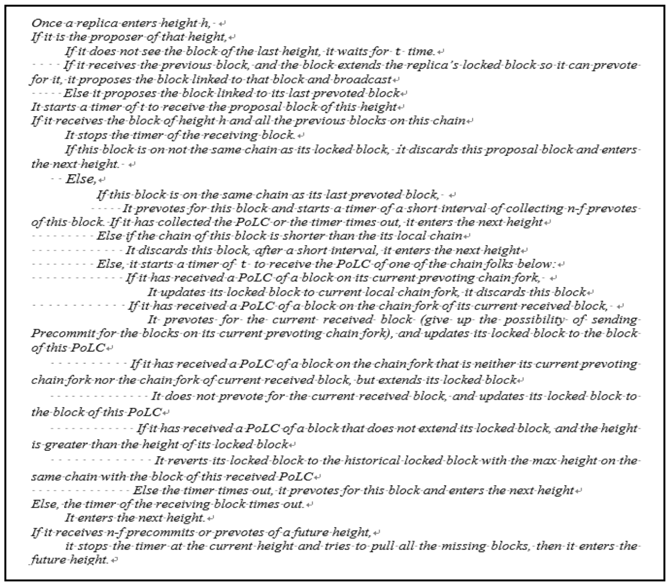
Now discuss the impact of the change of pace on liveness:

Since there is no strict constraint of conditions of entering the next height, we set up two timeouts for each height, one is for receiving the proposal block and the other one is for collecting *Prevotes* of current height, if one of these two times out, the replica is allowed to enter the next height. Then, if it is the block proposer of the next height, it can make proposal block and broadcast, otherwise it begins receiving the proposal block. We should allow proposal block skips blocks, which means the block of height n can link to block $n-2$ or even earlier one. This causes the possibility of chain forks even if all replicas are honest replicas.

We now discuss what the replica should do when he receives a proposal block on the other chain fork with his last prevoted block, To better discuss this, we assume the network model is: every message in the network should be received by every replica in T time with a very high probability.

1) If the received proposal block is not on the same chain with his locked block, he simply discards it.

2) Else, he faces a choice, if he chooses to prevote the block on another chain, he loses the opportunity to send *Precommit* for its previously prevoted blocks if he later collects PoLCs for them. So he should wait for the possible PoLCs to determine its action, there are four types of possible PoLCs may occur, and his corresponding action is stated in Fig. 2.



```
Once a replica enters height h,
If it is the proposer of that height,
    If it does not see the block of the last height, it waits for t time.
    If it receives the previous block, and the block extends the replica's locked block so it can prevote for it, it proposes the block linked to that block and broadcast
    Else it proposes the block linked to its last prevoted block
It starts a timer of t to receive the proposal block of this height
If it receives the block of height h and all the previous blocks on this chain
    It stops the timer of the receiving block.
    If this block is on not the same chain as its locked block, it discards this proposal block and enters the next height.
    Else,
        If this block is on the same chain as its last prevoted block,
            It prevotes for this block and starts a timer of a short interval of collecting n-f prevotes of this block. If it has collected the PoLC or the timer times out, it enters the next height
        Else if the chain of this block is shorter than the its local chain
            It discards this block, after a short interval, it enters the next height
        Else, it starts a timer of t to receive the PoLC of one of the chain folks below:
            If it has received a PoLC of a block on its current prevoting chain fork,
                It updates its locked block to current local chain fork, it discards this block
            If it has received a PoLC of a block on the chain fork of its current received block,
                It prevotes for the current received block (give up the possibility of sending Precommit for the blocks on its current prevoting chain fork), and updates its locked block to the block of this PoLC
            If it has received a PoLC of a block on the chain fork that is neither its current prevoting chain fork nor the chain fork of current received block, but extends its locked block
                It does not prevote for the current received block, and updates its locked block to the block of this PoLC
            If it has received a PoLC of a block that does not extend its locked block, and the height is greater than the height of its locked block
                It reverts its locked block to the historical locked block with the max height on the same chain with the block of this received PoLC
            Else the timer times out, it prevotes for this block and enters the next height
Else, the timer of the receiving block times out.
    It enters the next height.
If it receives n-f precommits or prevotes of a future height,
    it stops the timer at the current height and tries to pull all the missing blocks, then it enters the future height.
```

Fig. 2. The pace and the vote strategy of replicas in Parallel Tendermint

If set the timeouts of receiving the proposal block and the PoLC of it to t, $t < T$. We then discuss the replica should receive PoLC of height $n - x$ before the timeout of height n.

After a GST (Global Stable time) of height $n - 1$, every replica commits block $n - 1$ and enter height h, so every replica should receive the block n in T and receive all the *Prevotes* before $2T$. The quickest replica is it receives the proposal block of each height immediately, thus it only spends t time waiting for the PoLC of each height. $xt \geq 2T$, if we set $x = 4$, which means every replica receives the PoLC of $n - 4$ before the timeout of height n, then $t = T/2$.

Theorem 3 (Theorem of Commit). The block of height n is committed by every honest replica before end of height $n + 5$, if we set the timeout of the two timers in each height to be *T/2*.

Proof: From the analysis above, given one more T time for broadcasting and receiving *Precommit*, the honest replicas have received all the *Precommits*.

*Theorem 4 (Theorem of Jumping Height)***.** If a replica receives a future PoLC, it is allowed to jump to that height directly. This happens for at most f honest replicas for a future height.

Proof: If there are x honest replicas enter height h from previous height $h - 1$, for PoLC of the first jumping replica collects, there are $x + f$ *Prevotes* and $x + f \geq n - f$ thus $x \geq n - 2f$, the total honest replicas are $n - f$, thus the number of jumping replicas is $n - f - x \leq f$.

Due to jumping height, not every block can get PoLC, also because If the chain forks, the voter does not prevote the block not on his locked block.

Every replica keeps a record of the time elapsed since it receives and prevotes for a proposal block, if $2T$ has passed and not collected a PoLC for it, it should gradually increase the timeout of timers at later heights to increase the likelihood that the block and the PoLC in higher rounds are received by more replicas. We adopt a similar mechanism to increase the timeouts of the receiving block and collecting votes. Timeout:

$$t = \frac{T}{2} + 2^{(\text{current height} - \text{height of its locked block} - 4)} * \Delta$$

,and incremental timeout doubles after each height if replica has waited for four heights and still does not collect PoLC for the block of $n - 4$.

B. Comparison with Tendermint

From analysis above, the commonalities and differences between Parallel Tendermint and Tendermint can be sumed up

1) Commonalities

a) The principles to guarantee safety are nearly identical. The most important rule that each honest replica should obey is that it should stick to its locked block until it receives a PoLC of greater round. In our protocol, it is the PoLC of greater height.

b) The principle of increasing the duration of timeouts to increase the possibility that the replica sees a PoLC at some height in time is common to the two.

c) The proof of liveness of Parallel Tendermint is nearly the same as that of Tendermint because a PoLC at a greater height eventually update all replicas' locked blocks, quietly or can send Precommit for it.

2) Differences

a) Parallel Tendermint does not require the replica to commit to the block of a previous height to enter the next height. It "flattens" the rounds in Tendermint into the heights. Accordingly, Parallel Tendermint loses instant finality. Moreover, Parallel Tendermint does not require the replica to send Precommit before entering the next height, it can later send it if the PoLC for the block of that height is still acceptable

b) Parallel Tendermint allow the replica to update its locked block to the block of the received PoLC of which the height is greater than its locked height and less than its last prevoted height. In Tendermint protocol, the replica can only unset its locked block to nil in this case.

c) As Parallel Tendermint loses instant finality, the replica in it may switch between different chain forks, that means it needs to support the mechanism to rollback its execution point and execute the blocks on the other chain fork.

d) Parallel Tendermint does not require a replica to prevote *nil* if it does not see a block in time, and does not require a replica to precommit *nil* if it does not see a PoLC in time. The nil vote in Tendermint is a necessity for pace because a replica is allowed to enter the next phase only if it collects $n - f$ votes (including nil votes) in a given round. If Parallel Tendermint adopts this mechanism, each replica needs to vote nil for the blocks not on its locked chain, this adds to the network load. We thus use timers and the jumping mechanism with the PoLCs of future heights to loosely control the paces of the replicas.

C. Analysis of Performance

Performance is compared to Tendermint in two situations, one is "multiple rounds" has occurred, the other is the normal case where Tendermint commits each block in only one round.

Define b as time of broadcasting and receiving the block, define c as time of sending vote and collecting $n - f$ votes, and define e as time of executing transactions of the block. In Parallel Tendermint, two phases of voting (prevote and precommit) are parallel with broadcast and execution of blocks.

If a height of the Tendermint protocol needs r rounds to finish, and the total time is $\mathrm{r} * (b + 2c) + \sum_{i=1}^{r-1} \Delta_i + e$,of the same time, Parallel Tendermint can commit $(\mathrm{r} * (b + 2c) + \sum_{i=1}^{r-1} \Delta_i + e)/(b + \sum_{i=1}^{r-1} \Delta_i + e)$ blocks, but the real result should be less than that because Parallel Tendermint commits one of the chain fork, and each fork cannot contain full blocks, if there are two competing chain forks and each fork contains half the blocks then the real result halves, anyway the result should be greater than one block, compared to Tendermint only commits one block in multiple rounds in a height.

The other situation is the normal situation, where Tenermint commit each block in only one round. With n blocks, Tendermint spends time of $n * (b + 2c + e + \Delta)$, while Parallel Tendermint spends $n * (b + e + \Delta)$, here Δ.means the short interval when the replica enters the next height.

IV. Optimized Parallel Tendermint Protocol

A. *Using Aggregation of Votes*

The key thought of the optimization is to let the voter tell the aggregator what his current locked block is, if there happens to be $n - f$ voters locking on the same block, the aggregator publishes the aggregated signature, then the locked block can be committed.

The block proposer of each height is responsible for the aggregating for votes sent to it. The proposer is selected by a predetermined pseudo-random sequence or VRF (Verifiable Random Function) that is either distributed generated by VSS (Verifiable Secret Sharing) protocol [9] or local generated and confirmed by consensus [5].

B. *Combining Two Types of Votes into One Message*

As shown in Table II, the vote structure can be viewed as combining *Prevote* and *Precommit* together. Vote is like an arrow drawn from replica's locked block to its current prevoting block. Below shown two cases of paces when the chain does not fork.

TABLE II. Structure of a Combined Vote Message

Field	**Type**	**Description**
SourceBlockHash	byte[]	Source block stands for locked block of voter when it sends this vote. It is equivalent to a precommit of this voter.
SourceBlockHeight	big.Int	
TargetBlockHash	byte[]	Target block is Prevote block. Source block and target block must be on same chain.
TargetBlockHeight	big.Int	
SignatureForSource	byte[]	Signature of {SourceBlockHash, SourceBlockHeight} from voter's private key
SignatureForTarget	byte[]	Signature of {TargetBlockHash, TargetBlockHeight} from voter's private key

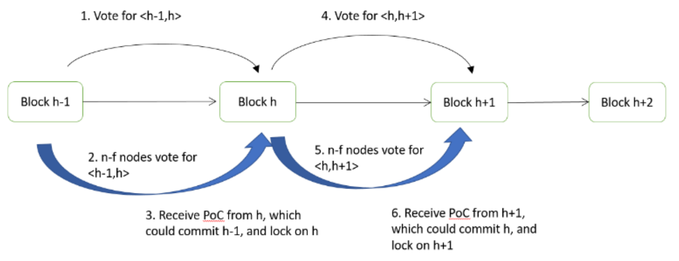

1. Vote for <h-1,h>
4. Vote for <h,h+1>
Block h-1
Block h
Block h+1
Block h+2
2. n-f nodes vote for <h-1,h>
5. n-f nodes vote for <h,h+1>
3. Receive PoC from h, which could commit h-1, and lock on h
6. Receive PoC from h+1, which could commit h, and lock on h+1

Fig. 3. The pace that replicas receive current PoLC to enter the next height

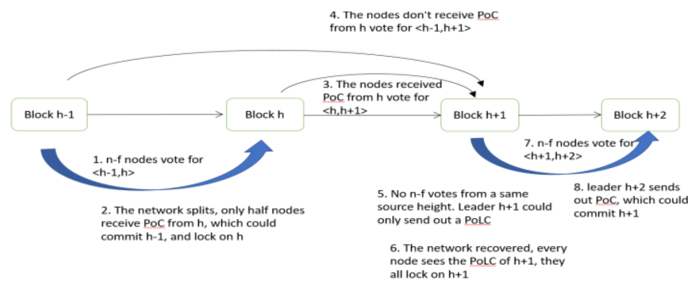

4. The nodes don't receive PoC from h vote for <h-1,h+1>
3. The nodes received PoC from h vote for <h,h+1>
Block h-1
Block h
Block h+1
Block h+2
1. n-f nodes vote for <h-1,h>
7. n-f nodes vote for <h+1,h+2>
2. The network splits, only half nodes receive PoC from h, which could commit h-1, and lock on h
5. No n-f votes from a same source height. Leader h+1 could only send out a PoLC
8. leader h+2 sends out PoC, which could commit h+1
6. The network recovered, every node sees the PoLC of h+1, they all lock on h+1

Fig. 4. The pace that replicas do not wait for current PoLC to enter next height

Even if a PoLC of a given height is not received by every replica in time, if the chain does not fork, every replica eventually agrees on extended block. In order to make this protocol work, we should adjust the voters' paces to make them locked on the same block as possible as they can. More implementation details are in §B. Below, we show an example which shows the protocol has the way to continue when the chain forks.

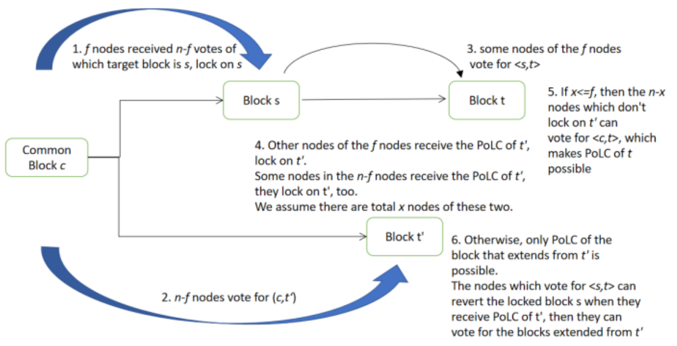

1. *f* nodes received *n-f* votes of which target block is *s*, lock on *s*
3. some nodes of the *f* nodes vote for <*s*,*t*>
Block s
Block t
5. If *x*<=*f*, then the *n-x* nodes which don't lock on *t'* can vote for <*c*,*t*>, which makes PoLC of *t* possible
Common Block *c*
4. Other nodes of the *f* nodes receive the PoLC of *t'*, lock on *t'*. Some nodes in the *n-f* nodes receive the PoLC of *t'*, they lock on t', too. We assume there are total *x* nodes of these two.
Block t'
6. Otherwise, only PoLC of the block that extends from *t'* is possible. The nodes which vote for <*s*,*t*> can revert the locked block s when they receive PoLC of t', then they can vote for the blocks extended from *t'*
2. *n-f* nodes vote for (*c*,*t'*)

Fig. 5. An example of liveness when the chain forks

V. Chained Tendermint Protocol

A. *Using Aggregation of Votes*

Instead of letting the voters tell the aggregator what their current locked blocks are, the Chained Tendermint Protocol gives them a one of two choice.

First, similar to HotStuff, the protocol lets each proposal block contain the PoLC of proposer's current locked block, we use same terminology as HotStuff, call it LockQC.

Second, this protocol uses one generic type of vote to express two meanings: a *Prevote* for this proposal block and a *Precommit* for thisProposalBlock.LockQC.block.

As the analysis in the basic Parallel Tendermint Protocol, sometimes the replica can prevote the received proposal block, but the LockQC contained in this block is not acceptable, thus it cannot send *Precommit* for it. One example is that the proposer's locked block is much earlier, and the voter has prevoted for blocks on other chain between the height of the LockQC.block and his locked height. For this case, we need to add an extra vote type "only *Prevote*", this type of vote differs from the generic type as it does not stand for the *Precommit* for LockQC.block. The vote format is shown below:

TABLE III. Structure of a Single Vote Message in Chained Mode

Field	**Type**	**Desription**
BlockHash	byte[]	Hash of the proposal block
BlockHeight	bigInt	Height of the proposal block
VoteType	bool	False means it is only a *Prevote* for the proposal block; true means it is also a *Precommit* for LockQC.block
Signature	byte[]	The signature of {BlockHash,BlockHeight, VoteType} from the voter's private key

Using this mechanism, each voter does not need to tell what its current locked block is, but only need to tell the proposer whether it can accept his LockQC or not, it is equivalent to it sends a *Precommit* to LockQC.block or not.

Since this is just a one of two choice, the possibility of there being $n-f$ *Precommits* of the same source height is significantly increased. We define a new vote message (shown in Table III) that adds a new field "VoteType" to indicate whether a voter prevotes for the target block and precommits for LockQC.block, or simply prevotes for the target block and does not precommit for LockQC.block. If there are fewer than $n-f$ voters that can precommit for LockQC.block, the aggregator can only generate the PoLC for the proposal block .

If we cancel the "only *Prevote*" vote type and do not allow the replica to prevote the proposal block if it cannot precommit for LockQC.block, this can cause the liveness problem. We give an example of this in §C.

Because the VoteType can be true of false, the voters may sign on two messages at each height. If there are not enough votes of "VoteType==true", the proposer has to aggregate the PoLC from two types of vote messages. Below are the description of signature aggregation and verification processes based on BLS in the cases of one message and two messages:

Signature aggregation. Given triples (pk_i, m_i, σ_i) for $i = 1,\ldots,n$ anyone can aggregate signatures $\sigma_1,\ldots,\sigma_n \in G_0$ into a short aggregate signature σ by computing $\sigma \leftarrow \sigma_1,\ldots,\sigma_n \in G_0$.

When all messages are the identical ($m_1=\ldots=m_n$), the verification process is a simple test that requires only two pairings: $e(g_1,\sigma) = e(pk_1 \ldots pk_n, H_0(m_1))$, where e is a non-degenerate, efficiently computable, bilinear pairing [7][10]

When there are two different messages (m_1 and m_2, suppose the signers $[1,i]$ sign on $m_1$and the signers $[i+1,n]$ sign on message m_2. The verification process is $e(g_1,\sigma) = e(pk_1 \ldots pk_i, H_0(m_1))e(pk_{i+1} \ldots pk_n, H_0(m_2))$, and requires three pairings.

Since this method needs to aggregate related public keys, verifier has to know the indices of the two groups of voters to verify aggregated signature. So we add two fields "IndicesOfPrecommit-Voters" and "IndicesOfPrevoteVoters" to indicate two groups.

TABLE IV. STRUCTURE OF A GROUP VOTE MESSAGE IN CHAINED MODE

Heading level	**Example**	**Description**
GroupSign	byte[]	Group signature
BlockHash	byte[]	Hash of the proposal block
BlockHeight	bigInt	Height of the proposal block
IndicesOfPrecommitVoters	uint[]	Indices of voters who both prevote the proposal block and precommit LockQC.block
IndicesOfPrevoteVoters	uint[]	Indices of voters who only prevote the proposal block

We hope the "two messages to form a PoLC" be a minor case because it adds complexity to aggregate and verify group signature and break the pipeline. If the network is good and there is no Byzantine behavior, the chain should not fork; then, each voter should accept the LockQC in each proposal block, in this case the "two messages to form a PoLC" should be minor and the protocol is still pipelined.

The process of handling a received block with a LockQC differs with Fig. 2 only in the following:

1) The replica should firstly extract the LockQC from the received block and handle it using the logic in Table I, this LockQC may update or revert its locked block. Based on the new locked block, the replica then judge whether it can accept this block using the logic in Fig. 2.

2) If the LockQC is a "one-message PoLC," as the VoteTypes of these votes are all true, this PoLC serves as a PoLC for *currentReceivedBlock.LockQC.Block* and a PoC for *currentReceivedBlock.LockQC.Block.LockQC.Block* which means the $n-f$ voters have all precommitted for it. In this case, it can be committed.

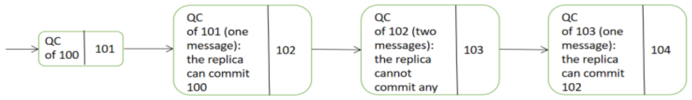


Fig. 6. The sequential commit in chained mode

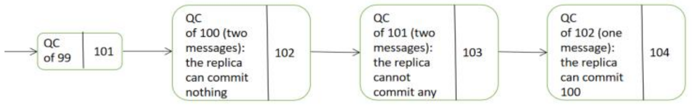


Fig. 7. The skipped height commit in chained mode

From Fig. 7, we can see, in the description of "*currentReceivedBlock.LockQC.Block.LockQC.Block*", the first LockQC must be "one message". As for the second LockQC, it can be "one message" or "two messages", in either case we can commit the block of the second LockQC.

In implementation, the leader can first broadcast a PoLC containing two messages, and can later broadcast a PoLC with only one message when it has collected $n-f$ votes with VoteType=true.

VI. CONCLUSION

The contributions of this paper can be summarized to:

1. We propose a basic parallel Tendermint protocol, which is a parallel version of Tendermint protocol, this protocol differs with Tendermint only in the paces of voting, but keeps the safety rules unchanged.

2. .There are three advantages of the chained Tendermint protocol to the chained HotStuff protocol.

   First, the phases of each block to be committed are reduced from three to two, thus each block needs less time to be confirmed on blockchain.

   Second, Chained Tendermint protocol does not need the "direct parent" constraint for the block to be committed, thus it is easier for some block in two competing forks meets the requirement of committing.

   Third, Chained Tendermint protocol does not need "view change" mechanism, it does not require the replicas send their highest QC to the next block proposer, each block proposer just attaches its local LockedQC to its proposal block. Thus the network load is reduced.

   More details of the comparison to HotStuff can be found in §D.

APPENDIX

A. An Example Showing the Acceptable PoLCs

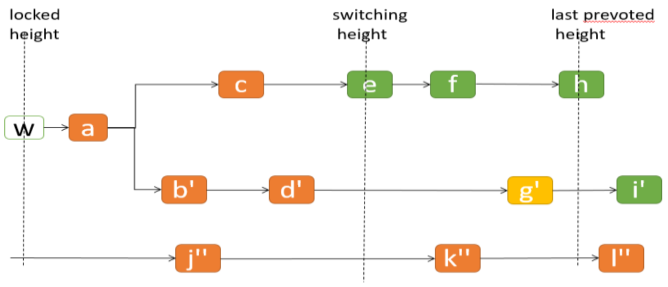


Fig. 8. The relationship between switch height and acceptable PoLCs"

This is a complex example, the replica locks on w, and prevotes for a, b', c, d', e, f, h. This is just for demonstration, Normally the replica does not change the prevoting chain so frequently. Now suppose that a, b', c, d', e, f, h, g', i', j'', k'', l'' all get $n-f$ *Prevotes*. Which of them are acceptable? PoLCs for a, b', c and d' are not acceptable because their heights are lower than the switching height of the replica; PoLC for g' is not acceptable because it is on a different chain with prevoting chain and its height is lower than height of last prevoted block h; PoLC for i' and l'' are acceptable because its height is greater than last prevoted height, they can update the locked block; PoLCs for j'' and k'' are not acceptable for their heights are lower than the last prevoted block h; PoLCs for e, f and h is acceptable because they are on its locked chain and their heights are greater than or equal to switching height.

B. Implementation Details of Optimized Parallel Tendermint Protocol

The aggregator of each height can collect $n-f$ votes from the same source height, and can aggregate two group signatures: a proof of commit for the source height, and a proof of lock for the target height.

When the $n-f$ votes are not from the same source height, the leader can only aggregate a PoLC for the target block. In this situation, the leader can also aggregate votes for the source block by source height. If there are m source heights, there are m group signatures. Note that votes for the source height should be ignored before last committed height. The leader sends these m group signatures to m leaders of the source heights. We call each of these group signatures, the size of which is smaller than $n-f$, the "part proof of commit" (PPoC). Each leader of each source height receives PPoCs from different leaders of the target heights. It can aggregate them if the total voter size of the PPoCs is greater than or equal to $n-f$; thus, a PoC is formed. Then, the leader of the source height broadcasts PoC.

TABLE V. THE THREE TYPES OF GROUP VOTE

Field	Type	Desription
Type	Enum (PoLC/PPoC/PoC)	Type of group signature
BlockHash	Byte[]	Hash of the block that this proof is for
BlockHeight	big.Int	Height of the block that this proof is for
Group Signature	Byte[]	Group signature

Note that regardless of whether it is a PoLC, PPoC, or PoC, the message should carry a field that indicates the indices of the voters so that the recipient can fetch their coordinate public keys to verify the group vote. In the process of aggregation of the PoC, it is possible that the PPoCs from multiple aggregators contains the same signature, thus the final PoC may contain duplicate signer indices.

C. The Liveness Problem If We Abandon "only Prevote" in the Chained Protocol

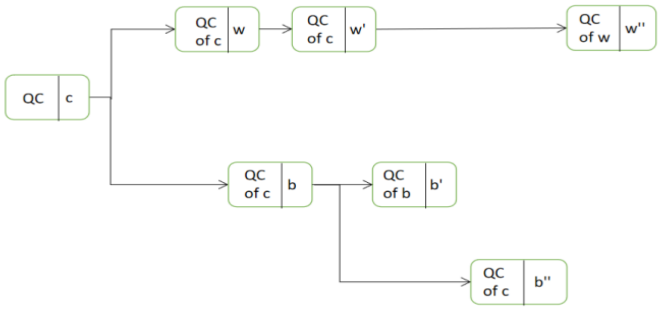


Fig. 9. The liveness problem if we abandon "only Prevote"

We assume that $n-f$ replicas prevote w but only f replicas see it. Thus they lock on w; the remaining $n-f$ replicas (including $f+1$ honest replicas and f Byzantine replicas) vote for block b. Of the $f+1$ honest replicas, some see $n-f$ votes for b and lock on it. Others do not, and can vote for w'. Suppose the f honest replicas locking on w all vote for w' and the f Byzantine replicas all do not vote for w'. Then $n-f$ votes for w' cannot be formed. The subsequent block w'' can thus only carry the QC of w. $f+1$ honest replicas that have voted for b cannot vote for it because the vote for w'' contains meaning of precommit w, which is contradictory to their prevotes for b.

We now consider blocks that extend b. Either replicas that first lock on w and then vote for w', or those that first vote for b and then for w' cannot vote for b'. This is because the vote for b' contains the meaning of precommit b, which is contradictory to their prevotes for w'. Only fewer than $f+1$ replicas that lock on b can vote for b'; even with the help of the Byzantine replicas, $n-f$ votes are not available. Block b'' with a QC of c can get $2f+1$ votes because the f replicas locking on w can unlock when they see the QC of . But the block proposer naturally picks the highest QC it has seen, and links the last prevoted block as the parent block. Replicas locked on b contain the LockQC of b in their proposal blocks, and replicas that have voted for w' propose a block that extends w'. We cannot let the block proposer arbitrarily choose another QC at a lower height or link to another parent block at a lower height to ensure liveness.

D. Related Work

We compared the proposed protocols with several BFT consensus protocols that are also without instant finality.

1) Comparison with Casper

Casper [3] can be viewed as akin to Optimized Parallel Tendermint. The two rules of Casper does not follow the safety requirement of Tendermint, it allows the two votes ⟨s1,t1⟩ and ⟨s2,t2⟩, where s1<s2<t1<t2 and s1,t1 are on different chains from s2,t2. If we treat s1 and s2 as *Precommits*, and t1 and t2 as *Prevotes*, Optimized Parallel Tendermint does not allow these votes. A stricter voting rule leads to a looser commit rule, whereas Casper needs two sequential $n-f$ votes to commit a checkpoint (similar to the constraint of "direct parent" in HotStuff). Our solution does not need this constraint. Stricter vote rule causes the replicas to more quickly switch to the chain fork with the greatest PoLC. On the other hand, the plausible liveness in Casper is weak since there is no unlock mechanism, this causes the replica very difficult to switch to another chain fork on some cases, e.g. when the replica has voted for <h, h+1>, any vote on another chain fork that covers the range violates either rule 1 or rule 2.

2) Comparison with HotStuff

The basic HotStuff requires the extra prepare phase and the mechanism that every replica send its highest PrepareQC to the leader of the new height to ensure liveness. The proof is: if a replica has locked to some block *b*, there should be $n-f$ *Prepares* for it, so there should be at least $f+1$ honest replicas that have seen the PrepareQC, they will send it to the new leader, and if the new leader is non-faulty, it will propose a block linked to the block of this PrepareQC, so the later blocks should extend the locked block. From the author's point of view, there is a liveness issue: what if the new leader is faulty and it proposes a block on another chain fork that does not extend the locked block? Though it can only provide a lower PrepareQC for this, the voters except the one locked to *b* can still vote for it, because the lower PrepareQC is higher than their locked height. Then this block can be committed. There is no unlock mechanism in HotStuff, and the replica accepts only the block which extends its locked block, so the replica which locks on *b* become forever blocked. A question is: since there is no unlock mechanism, why doesn't the replica commit the block directly when it needs to lock to it?

One of the advantages of our chained protocol to chained HotStuff is that our chained protocol does not need the "direct parent" constraint. This is because our chained protocol still follows the safety rules of Tendermint while chained HotStuff does not. Why chained HotStuff needs the additional constraint "direct parent" while the basic HotStuff doesn't? In the chained model. the generic vote carries multiple meanings, e.g. *b*→*b'*→*b''*→*b**, the vote for *b** can stand for the *Prepare* for *b** and the *Prevote* for *b''* and the *Precommit* for *b'* and the *n-f Precommits* for *b* thus *b* can be committed. Thus it breaks the height monotonicity, in other word, the votes for the blocks of different heights are parallelized, so it is possible for a replica to prepare for some height *h*, and later precommit for some height *h'* that *h'*<*h*. Since the chained protocol does not consider to keep consistency of the replica behavior in the basic protocol and the chained protocol, it has to setup the "direct parent" constraint so that *b, b'* and *b''* should be strictly sequential, no votes for other blocks on other fork can be between them (see Fig. 10)

The mechanism to let all replicas send their highest PrepareQC to the new leader adds extra network load, in our protocol, we simply let the proposer attach his own PoLC of his current locked block as the LockQC in his proposal block.

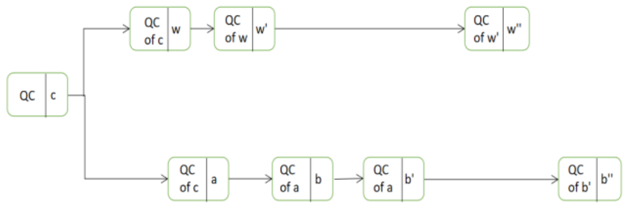


Fig. 10. Example showing necessity of constraint of direct parent in HotStuff

A replica can vote for *w* and *a*, and can then vote for *w'*. It can lock on *w* and discover *b*, where the height of the QC of a is greater than that of *w*. It can thus can vote for *b* and lock on *a*. The block b' does not contain the QC for b because we assume there are not $n-f$ votes for b. Because b' is on the same chain as its locked block a, it can still vote for b'. Then, block w'' with a QC for w' appears, where the height of w' is greater than the height of a. It can thus vote for w'' and lock on w'. Then, block b'' with a QC of b' appears, where the height of b' is greater than the height of w'. It can thus vote for b'' and lock on b'. Then, both w'' and b'' can gather $n-f$ votes and both chain forks can be committed.

In the protocols of this paper, the replica should follow Theorem of Tendermint strictly, not only in case it first locks on some block and does not prevote the blocks on other chain forks, but also in the case that it first prevotes for some block and is not allowed to send *Precommit* to the block at a previous height on another chain fork. In Fig. 10, because the replica has prevoted for *a*, it cannot vote for *w'*, even if the LockQC of w in *w'* indicates that its height is greater than the locked height of block *c*. If the replica receives $n-f$ *Prevotes* for *w*, it cannot lock on *w* either because it has prevoted for *a*, which is on a different chain and has a height greater than the height of the received LockQC.

3) Comparison with Grandpa

In Grandpa protocol, there are also two steps — prevote and precommit — in a round, but the Precommit is not the commitment for a locked block, but a declare for $g(V_{r,v})$ — the common block of the prevote set of round *r* with greatest height. The estimated block $E_{r,v}$ is extended from $g(V_{r,v})$ and possible to be $g(C_{r,v})$. It declines as the replica *v* sees more Precommits and judge that some descendant block of $g(V_{r,v})$ is impossible. If we view the estimated block as the temporary locked block of a replica, as more votes it receives, temporary locked block declines rather than increases. While in Tendermint, Casper, or HotStuff, if the locked block of a replica gets updated, the height of the locked block must increase. This difference causes Grandpa challenging to convert to a chained model.

The ghost function $g()$ is to get a greatest common block from a vote set, the votes are for different blocks possibly on different chain forks. Thus it is challenging to use aggregating signature to reflect the vote set and the result.

As for the fork choices, for the replica votes for the block on the longest chain, thus if there are several chains forks and the longest chain varies between them, the $g\left(V_{r,v}\right)$ may not get increased. Perhaps it needs a similar gradually increasing timeout as (the height of current voted block - the height of last committed block) increases, but the paper doesn't mention it.

E. Why We Cannot Commit the Block with the Minimal Height in $\boldsymbol{n-f}$ Precommits for Different Blocks on the Same Chain

The reason is: the replica which sends the Precommit can change its locked block in later heights, so another $n-f$ Precommits for the other block could exist. Only the $n-f$ Precommits of a same height for same block can infer the fact that there are not $n-f$ *Prevotes* for any other block at later heights on another chain according to Theorem 2.

References

[1] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. arXiv preprint arXiv:1807.04938, 2018. https://arxiv.org/abs/1807.04938.

[2] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT Consensus in the Lens of Blockchain. CoRR abs/1803.05069 (2018). arXiv:1803.05069 https://people.csail.mit.edu/nickolai/papers/gilad-algorand-eprint.pdf.

[3] Vlad Zamfir. Casper the friendly ghost: A "correct-by-construction" blockchain consensus protocol, 2017. arXiv:1710.09437. https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf

[4] Alistair Stewart. Grandpa Byzantine Finality Gadgets. https://github.com/w3f/consensus/blob/master/pdf/grandpa.pdf.

[5] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. ACM, 2017, pp. 51-68. https://people.csail.mit.edu/ickolai/papers/gilad-algorand-eprint.pdf.

[6] M.pease, R.Shostak, L.Lamport. Reaching agreement in the presence of faults, J.Assoc.Comput. Mach.27, No.2, 228-234

[7] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. In Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology, ASIACRYPT '01, pages 514–532, London, UK, UK, 2001. Springer-Verlag.

[8] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA.

[9] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. DFINITY Technology Overview Series, Consensus System. CoRR abs/1805.04548 (2018). arXiv:1805.04548.

[10] D. Boneh, C. Gentry, H. Shacham, and B. Lynn, "Aggregate and verifiably encrypted signatures from bilinear maps," in Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRPYT '03), vol. 2656 of Lecture Notes in Computer Science, pp. 416–432, Springer, Warsaw, Poland, May 2003.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, Impossibility of distributed consensus with one faulty process. No. MIT/LCS/TR-282. Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982

[12] L.Lamport, R.Shostak, M.pease. The Byzantine Generals problem, ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, July 1982, pp. 382-401.