Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil

Ittai Abraham, Guy Gueta, Dahlia Malkhi
VMware Research

March 13, 2018

Abstract

We describe a protocol called 'Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil' (in short, Hot-Stuff) for $n = 3f+1$ replicas, of which $2f+1$ are honest, to agree on a replicated, ever-changing state. The protocol is always safe against a threshold f of Byzantine failures, even when the system is asynchronous. Progress is guaranteed under periods of synchrony. The per-round communication cost in Hot-Stuff is linear, hence $O(n^2)$ overall cost to a decision during periods of synchrony, an improvement of $O(n^2)$ over previous asynchronous BFT protocols. Hot-Stuff uses one type of message exchange, and is succinctly described in under twenty lines of pseudo-code.

1 Introduction

We describe a protocol called 'Hot-Stuff the Linear, Optimal-Resilience One-Message BFT Devil' (in short, Hot-Stuff) for $n = 3f + 1$ replicas, of which $2f + 1$ are honest, to agree on a replicated, ever-changing state. The protocol is always safe against a threshold f of Byzantine failures, even when the system is asynchronous. Progress is guaranteed under periods of synchrony. Solving state replication in this settings is henceforth referred to as the *BFT problem*.

In the same settings, Dwork et al. demonstrated in [22] the existence of a safe solution for Byzantine consensus. However, the solution was not designed for efficiency. Even under the best possible conditions, where the system behaves synchronously from the outset, and there are no failures, completion requires $O(n^4)$ signed values to be transmitted over a wire. Briefly, the complexity stems from the DLS *view-change* protocol for transitioning from one *proposer* to the next. In order for a new proposer to select a safe value to propose and for the proposal to be accepted, each view-change consists of an all-to-all exchange among the replicas of *proofs* about proposals. The size of each proof is $O(n)$, hence the total communication complexity of each view-change instance is $O(n^3)$, and there is a sequence of $O(n)$ view-changes toward a decision.

More than a decade later, Castro and Liskov introduced a solution called PBFT [15] that optimizes for the case of a stable, fail-free proposer. This was followed by a line of works that perform speculatively fast under fail-free conditions [26, 28, 25, 17, 5, 23]. Among these works, both PBFT and several others extend the treatment to State-Machine-Replication (SMR), forming agreement over a growing log of commands, and applying them to replicate state.

Despite all this progress, to this date, the fundamental complexity of $O(n^3)$ transmissions associated with handling a proposer failure through a view-change remains. Consequently, all the above works suffer $O(n^4)$ transmissions associated with handling a cascade of f failures. These complexities arise in these works even under completely synchronous conditions. One might argue that this is not so bad, as steady state is all we care about, and an occasional performance hit is fine when coping with failures. However, as discussed in several works [4, 17, 19, 18, 6], proposer-rotation may be employed even without detecting a failure in order to promote fairness and protect against undetectable performance degradation attacks. Additionally, for very moderate systems sizes, e.g., $n = 100$, being hit (even on occasion) with $100,000,000$ transmissions for a single consensus decision is a scalability barrier.

Recent advances in Byzantine fault tolerance that emerged in the arena of decentralized cryptocurrencies [10, 12] have very different "feel" than PBFT and its variants: They progress by publishing a proposal with a single value, they have no explicit view-change, and their proposals carry no explicit proof of safety. Inspired by these works, we introduce Hot-Stuff, a BFT protocol whose per-round cost is linear, and overall cost per decision $O(n^2)$, thus an improvement of $O(n^2)$ over previous BFT protocols. Hot-Stuff uses one type of message exchange, and is succinctly described in under twenty lines of pseudo-code (see Algorithm 1). It provides a rigorous algorithmic foundation and proof of correctness for protocols like Tendermint [10] and Casper [12], as well as $O(n)$ improvement over all known solutions in communication complexity. Our work embodies the following contributions.

Linear View-Change: Our first contribution is a *Linear View-Change* (LVC) scheme for driving proposer-replacement, that reduces by factor $O(n)$ the communication complexity in PBFT. Accomplishing the linear reduction in LVC does not result in a complex protocol, but on the contrary, simplifies PBFT considerably. This lends LVC to robust implementations, and may be instrumental for teaching. We provide an overview of the LVC/PBFT solution in §3. We remark that the LVC approach, by itself, can be applied to protocols that have a speculative fast track [28, 25, 18, 5, 23], thus getting the benefits of a linear view-change on top of the benefits of a fast track.

Single Index: Our second contribution is a single-index SMR solution framework. Instead of growing in two dimensions–the rotation of proposers and the sequence of commands–we unify the two dimensions into one. A single growing index indicates both a command slot and a proposer phase. We refer to the single dimension as a *level*.

Classical BFT solutions, like PBFT [15] and Zyzzyva [25], rotate proposers within each sequence slot, and separately, advance slots when decisions are reached. Traditional works in benign SMR settings, like VR [30], Paxos [27], ZaB [24] and Raft [31], take a similar approach. This is so confusing that it has reportedly led to bugs in published works and in deployed commercial systems alike: Zyzzyva suffers from a safety issue because it uses the log-length where view-numbers should be used [2], FaB gets stuck in scenarios in which view-numbers would get it unstuck [2], vSAN used to get stuck because it embedded configuration changes into slots and used slot-numbers also as view-numbers [9], and Tendermint suffers a liveness issue related to conflicting proposals [14].

A single-index approach is cleaner and easier from a pedagogical point of view. Additionally, it is compatible with current BFT use-cases in distributed blockchains, where there is no explicit proposer phase, only the chain length it proposes.

Single Message: We further improve the overall solution via a single message framework, a PROPOSE by a *beacon* (see below) and VOTE responses by replicas. First, Hot-Stuff completely removes proposer-replacement messages, and treats every proposer as a new one. Second, the proposer's protocol in Hot-Stuff has only a single phase. In PBFT [15] there are two exchanges, PREPARE and COMMIT, and likewise, a two-phase paradigm is employed in the recovery-track of follow-on protocols with speculatively fast tracks [28, 25, 18, 5, 23]. Hot-Stuff coalesces the two steps into a single exchange by pipelining the COMMIT step into the PREPARE step of the next level. This results in only a simplified voting framework, and in efficient pipelining.

Signature Combining: Threshold signatures have proved useful in previous BFT systems (e.g., [32, 33]). We explain in §4 how to obtain further reduction of factor $O(n)$ in communication complexity by using threshold signatures. This brings the communication cost down to linear per level, hence $O(n^2)$ overall cost to a decision during periods of synchrony, which is optimal even in purely synchronous settings [21]. Hence, Hot-Stuff is an asynchronous protocol that incurs essentially no cost over synchronous protocols, despite maintaining safety during periods of asynchrony.

Beacon: We present the Hot-Stuff framework using an abstract *beacon* functionality, a mechanism and policy for triggering proposals. The beacon encapsulates a "Longest-Fork-Lives" rule, requiring that proposals be consistent with the highest commit candidate of Hot-Stuff infinitely often. The abstraction allows for different implementations to be plugged in. In particular, we explain in §7 how a beacon may be realized by a Proof-of-Work network completely auxiliary to the replica set. This concrete instantiation of Hot-Stuff may be utilized as a *finalizing gadget* [11] in Proof-of-Work settings, and we name it *Wendy the View-Changing Finality Gadget*.

2 Background

Model. For most of the paper, we consider a system consisting of a fixed set of $n = 3f + 1$ *replicas*, of which up to f are Byzantine faulty, and the remaining ones correct. We adopt the celebrated *partial synchrony model* of Dwork et al. [22], where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two correct replicas arrive within time Δ. In this model, progress will be guaranteed within a bounded duration after GST. In practice, the GST model is sufficient to guarantee progress if the system alternates between periods of asynchrony and (sufficiently long) periods of synchrony, but simplifies the discussion. We refer to the problem of maintaining replicated ever-changing state in this model as the *BFT problem*.

Crypto. We make use of standard cryptographic signatures to sign messages. Additionally, we utilize threshold signatures [33, 13, 7], where for a threshold parameter k, any subset of k from a total of n signers can collaborate to produce a valid signature on any given message, but no subset of less than k can do so.

We assume a PKI setup between replicas for authentication and for setting up a $(2f+1)$-out-of-n signature scheme.

Complexities. The key complexity measures we care about is communication *bit complexity*. Bit complexity measures the number of bits transmitted over any wire. This measure hides unnecessary details about the transmission topology, and makes it possible to compare "apples to apples": For example, n messages carrying one value count the same as one message carrying all n values. As another example, an all-to-all exchange of values counts the same when messages are collected by a "relay" node and disseminated to others as when they are sent directly. To make it more palatable, we ignore low-level protocol headers, acknowledgements and the like, and regard the transmission of $X \times \max\{polylog(n), sec\}$ bits, where sec is a security parameter such as the number of bits in a signature, as $O(X)$.

DLS. The DLS solution [22] works in a sequence of *phases*[1]. In each phase, there is a dedicated proposer trying to obtain $2f+1$ replicas to *lock* a proposal. If it succeeds, the proposal becomes the committed decision. No replica will *unlock* unless it receives a *proof* that a higher phase obtains $2f+1$ replicas to lock a different value. Since this requires $f+1$ correct replicas to already be unlocked, it cannot happen. In order to provide liveness, every DLS phase ends with an unlock round in which replicas broadcast to everyone the locks they hold, each with a corresponding proof. Therefore, every phase entails a transmission of $O(n^3)$ signed values. After some decision value has been fixed, one by one proposers must learn that it is the only safe value to propose, send it again as a proposal, and receive $2f+1$ votes for it, for a total of $\Omega(n)$ phases.

PBFT. PBFT [15] extends the treatment to solve the State-Machine-Replication (SMR) problem. Under conditions of synchrony and no proposer-failure, PBFT can complete a consensus decision on each command slot with $O(n^2)$ communication complexity. However, when faced with up to f failures, even under synchronous conditions, it may require $O(n^4)$ communication to reach one decision.

More specifically, PBFT forms a succession of consensus decisions, one per slot. The consensus solution for each command slot works in a sequence of phases (called views in [15]). In each phase, a dedicated proposer uses a two-step voting procedure, a PREPARE vote and COMMIT vote. A PREPARE-vote provides uniqueness, at most one proposal per phase obtains $2f+1$ votes. If a replica receives $2f+1$ PREPARE-votes for a proposal, it assembles them into a COMMIT-certificate, and sends a COMMIT-vote for it. A decision becomes committed by $2f+1$ COMMIT-votes. In ideal conditions, all replicas can learn a decision in one phase via an all-all COMMIT-vote exchange incurring $O(n^2)$ communication complexity.

A replica becomes "locked" on a value after it has assembled a COMMIT-certificate and sent a COMMIT-vote on it. Should a proposer fail, replicas enter a view-change procedure to exchange their COMMIT-certificates. This procedure fundamentally falls back to the DLS view-change scheme, except that it is driven by the new proposer collecting COMMIT-certificates and disseminating them. The communication complexity is $O(n^3)$, and a cascade of f proposer failures may cause this to recur $O(n)$ times.

3 LVC/PBFT

In this section, we shed insight into a modification to PBFT called LVC/PBFT, that reduces the overall communication complexity associated with a view-change to $O(n^3)$, hence a reduction by factor $O(n)$ over vanilla PBFT. We present a single consensus slot protocol only. The details concerning extending the solution to state-replication can be easily worked out by following the PBFT template. We also avoid a rigorous description of LVC/PBFT at this point, since we embed the same LVC principles within Hot-Stuff, for which we provide a full description in the next section.

The key idea behind LVC/PBFT is that a new proposer does not provide a "proof" from $2f+1$ replicas about the highest COMMIT-certificate they hold. Instead, in LVC/PBFT, a new proposer provides one

[1]Related notions in the literature to a phase are a ballot (Paxos [27]), a view (View-stamped Replication [30]), a round (Raft [31]), a height (Tendermint [10], Casper [12]), and other names.

COMMIT-certificate only. A faulty proposer can easily hide the highest certificate it has collected from replicas (including itself), if this serves its interests. To preserve safety nevertheless, we add the following rule: A correct replica unlocks a COMMIT-vote only if it receives a COMMIT-certificate from a higher phase. Therefore, intuitively by hiding the highest certificate a proposer simply risks being declined, but not damaging safety.

That's the entire change in a nutshell! We name the scheme *Linear View-Change*, or LVC, because a new proposer sends a message carrying only a linear number of signed values, as compared with vanilla PBFT, where a new proposer sends a quadratic number of signed values.

Just like vanilla PBFT, a further improvement in communication complexity is achievable via signature combining (threshold signatures) on COMMIT-certificates, as discussed in the next section. This enhancement results in a linear reduction throughout the protocol, wherever a message needs to carry a COMMIT-certificate. Importantly, this improvement works over the LVC/PBFT reduction, hence it results in an overall $O(n^2)$ view-change cost.

Protocol. We now proceed to sketch the entire LVC/PBFT protocol for a single consensus slot. The protocol operates in a sequence of phases. We assume some mechanism exists for advancing phases and guaranteeing progress during periods of synchrony, such as the ones discussed later in §6.

We first describe the protocol for a single phase (k), and later introduce the view-change protocol. A dedicated proposer per phase uses a two-step voting procedure, a PREPARE-vote step and COMMIT-vote step. For each slot, the proposer waits to collect command requests (from clients) until a fixed batch size is reached, or until a certain timeout period has expired. For simplicity, we will henceforth refer to the batch as a single command. The proposer sends the command in a signed PREPARE message to all the replicas.

When a replica receives a PREPARE message from the proposer of phase k it takes the following steps. If this is the first proposal of a new proposer then the message carries a COMMIT-certificate. In that case, if the replica holds a lock with a lower phase than the certificate, then it releases it. For every proposal (including the first), the replica *accepts* the proposal if either (i) the replica does not hold any lock, or (ii) it already holds a lock on the command that is being proposed. Upon accepting it, it sends a PREPARE-vote with a signed digest of the proposal to all other replicas.

When a replica receives $2f + 1$ PREPARE-votes for a phase-k proposal, it assembles them into a COMMIT-certificate. The replica sends a COMMIT-vote with a signed digest of the certificate, and becomes *locked* on it. As mentioned above, a replica unlocks a certificate only it it receives a COMMIT-certificate from a higher phase. If a replica receives a signed COMMIT-vote with a digest of an unknown certificate, it asks the sender to forward the certificate and then behaves in an identical manner to when it assembles the certificate itself.

A command becomes a *committed decision* when a COMMIT-certificate for the command receives $2f + 1$ COMMIT-votes.

When moving to a new phase, each replica sends to the new proposer a NEW-VIEW message carrying the COMMIT-certificate it is locked on, if any. The new proposer picks the highest COMMIT-certificate among $2f + 1$ NEW-VIEW messages it receives. In the first PREPARE message, the new proposer piggybacks this certificate. We already describe above how this PREPARE is handled by replicas.

Correctness. Briefly, the reason LVC/PBFT maintains safety is the follows. If a value becomes a committed decision in phase k, then there are $f + 1$ correct replicas locked on it. For a COMMIT-certificate with a different value to be formed at a higher phase, at least one of these replicas must become unlocked and vote PREPARE for it. For this vote to happen, there needs to already be a COMMIT-certificate with a different value at a higher phase. Hence, no such certificate can be formed, and hence no conflicting decision may be committed.

Liveness is guaranteed during periods of synchrony because a correct new proposer is guaranteed to receive the highest COMMIT-certificate held by any correct replica. Therefore, it will use in its first PREPARE some certificate that will be accepted by all correct replicas. Since votes are synchronous, a decision will be reached.

4 Hot-Stuff

Algorithm 1 Hot-Stuff BFT protocol

local variables

$proposals$: ▷ every known proposal
command($level$) × parent

$highCC, highTail$: ▷ highest-level COMMIT-certificate, highest-level tail extending it
command($highlevel$) × $(2f+1)$ votes , command($tail$);

SMR : ▷ state-machine-replication
state × highcommitted;

$curlevel$; ▷ initially 0

Replica functionality:

upon beacon-proposal $\langle \text{PROPOSE}, CC(cclevel), parent, newcmd(\ell)\rangle$:
(omitted for brevity: verify information, retrieve missing ancestors as needed)
if $cclevel > highlevel$ **then**
$highCC(highlevel) \Leftarrow CC(cclevel)$;
(omitted for brevity: apply new commits from $CC(cclevel)$ to SMR, if any)
update $highTail(tail)$ to highest-level command extending $highCC$; ▷ **safe-cc**
send $\langle \text{VOTE}, \ell, digest(highCC), digest(highTail)\rangle$ to $(\ell+1)$-proposer;

Beacon functionality (abstract):

send to all replicas:
$\langle \text{PROPOSE}, highCC(highlevel), highTail(tail), newcmd(\ell)\rangle$;

where inifinitely often proposal satisfies:
$\ell > curlevel$,
$newcmd$ extends highest COMMIT-certificate in the system,
at least 2Δ period elapses to next PROPOSE broadcast

Data-Structures

The Hot-Stuff algorithm revolves around one data-structure, a *log* consisting of a sequence of proposed commands. We think of the algorithm as committing the log *level* by *level*.

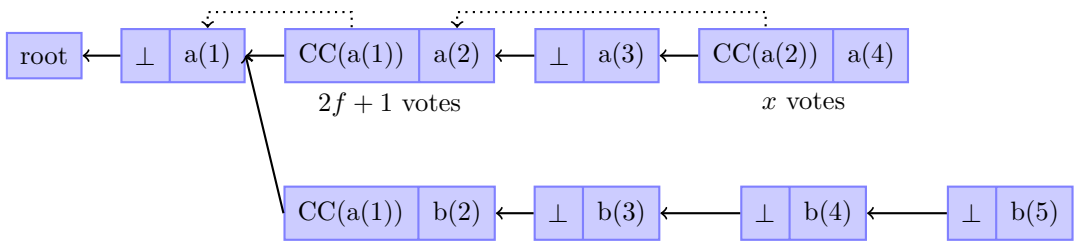
Each log entry contains a reference to (e.g., a cryptographic digest of) its *parent* entry in the log (as used in, e.g., Raft [31], Nakamoto Consensus [29], and other log replication schemes). A command at level ℓ unambiguously represents an entire *ancestry chain*, up to an origin called the *root*. Whenever we discuss a command *cmd* at level k, we denote it by $cmd(k)$.

Two commands are *conflicting* if one is not a prefix of the other. If there are multiple, conflicting proposals at the same level, a replica keeps all of them until one becomes committed, and then it discards all conflicting prefixes.

A key notion used in the scheme is a COMMIT-certificate. Only a unique command per level may obtain a COMMIT-certificate. It consists of $2f+1$ votes on a command $cmd(level)$, and is denoted $CC(cmd(level))$.

A proposal may carry an optional COMMIT-certificate for some ancestor. A VOTE on a proposal carrying a certificate for a direct parent constitutes a COMMIT-vote. $2f + 1$ COMMIT-votes indicate a command and its prefix are *committed*.

Here is an example of the local information kept at a replica:



In this example, the first level has a unique proposal $a(1)$, and it obtains $2f+1$ votes. The CC for $a(1)$ is pushed into the next level. At level 2, a fork with two conficting proposals is created, $a(2)$, $b(2)$. This could be generated by a single bad proposer or by having proposer contention at level 2. Both of these proposals carry the CC for $a(1)$, but only one of them may obtain $2f + 1$ votes. In this case, $a(2)$ obtains $2f + 1$ votes, i.e., a CC. The votes on $a(2)$ also constitute COMMIT-votes on $a(1)$. Hence, the branch up to (incl.) $a(1)$ becomes committed. At level 3, a proposal is made without waiting for the CC on $a(2)$. Proposition $a(4)$ carries a CC for $a(2)$ which cannot become committed. The votes on $a(4)$ are collected, and in this case, so far x were obtained. The conflicting branch b continues in this example, but it cannot commit any conflicting decision.

Protocol

The entire Hot-Stuff protocol, including data-structures, is described within one frame in Algorithm 1.

It operates level by level, and encompasses two abstract roles, a **replica** role and a **beacon** role. A beacon for level m sends to all replicas a PROPOSE message. A replica reacts by sending a VOTE response back to the beacon for collecting. All messages are signed by their senders.

Beacon functionality. The beacon functionality is to collect a COMMIT-certificate per level, and push the COMMIT-certificate to the replicas for a vote. When a new level *curlevel* starts, the beacon advances the log tail to *curlevel* (padding the known tail of log with empty commands up to *curlevel*, as needed), and sends a proposal:

$$\langle \text{PROPOSE}, highCC, digest(cmd(curlevel - 1)), newcmd(curlevel) \rangle$$

A PROPOSE message carries the following information. It includes a proposal $newcmd(curlevel)$ extending the current tail of the log to *curlevel*. It carries a parent reference (e.g., a cryptographic digest) to the previous log tail $cmd(curlevel - 1)$.

The beacon collects votes from $curlevel - 1$ and tries to form a COMMIT-certificate. Safety does not depend on the beacon having a complete COMMIT-certificate for level $curlevel - 1$. The beacon is allowed to send a COMMIT-certificate for a lower level, or none at all. The proposal simply carries the highest COMMIT-certificate known to the beacon, denoted $highCC$.

Liveness depends on beacons waiting for sufficiently long to have infinitely many levels in which a COMMIT-certificate is formed. Several concrete realizations of the beacon functionality and vote collection are discussed in §6 and §7.

Replica functionality. The replica role is to maintain safety by voting on proposals. As discussed above, votes need to be collected by the beacon of the next level. Once a replica accepts a PROPOSE message, the replica sends the beacon a vote:

$$\langle \text{VOTE}, digest(proposal) \rangle$$

The following safety rule determines if the replica should accept a proposal:

safe-cc: A replica sends a vote for a command only if it does not conflict with the highest-level COMMIT-certificate prefix it saw (including the one carried in the PROPOSE message itself).

If *proposal* carries a COMMIT-certificate for a direct parent $CC(cmd(curlevel - 1))$, then the vote constitutes a COMMIT-vote on the COMMIT-certificate. Once $2f + 1$ COMMIT-votes are made, $cmd(curlevel - 1)$ becomes a *committed* decision.

Signature Combining

An advantage of the beacon functionality abstraction is that a proposer can collect and combine votes using threshold signatures. Every COMMIT-certificates is transformed in this manner into a single signature on a command. Thus, the overall communication complexity per round is linear.

5 Hot-Stuff Correctness

We denote an event $\langle \text{PREPARE} : cmd(j) \rangle_r$ to indicate that replica r votes for a PROPOSE carrying $cmd(j)$ as the new command. We denote an event $\langle \text{COMMIT} : cmd(j) \rangle_r$ to indicate that r votes for a PROPOSE carrying a COMMIT-certificate for a direct parent $cmd(j)$. We denote an event $\langle \text{PROPOSE} : CC(cmd(j)) \rangle$ indicating that a beacon proposal carries a COMMIT-certificate for $cmd(j)$.

Lemma 1. *Denote the following events:*

$$e_j = \langle \text{ COMMIT } : \ cmd(j) \ \rangle_r$$
$$e_k = \langle \text{ PREPARE } : \ cmd(k) \ \rangle_r$$
$$e_h = \langle \text{ PROPOSE } , \ CC(cmd(h)) \ \rangle$$

Suppose that events e_j and e_k occur (at replica r), such that $j < k$ and command $cmd(k)$ conflicts with $cmd(j)$. Then event e_h must exist, such that $j < h$, and $cmd(h)$ does not conflict with $cmd(k)$.

Proof. By assumption, event e_j must occur before e_k. Therefore, $CC(cmd(j))$ is already known to r when event e_k occurs. According to the **safe-cc** rule (see Algorithm 1), $CC(cmd(j))$ must not be the highest-level certificate known to replica r when e_k occurs; for otherwise, e_k could not be accepted by r. It follows that r receives before e_k occurs a COMMIT-certificate that overrides $CC(cmd(j)$. We denote it by $CC(cmd(h))$, and the lemma follows. □

Corollary 1. *Denote the following events:*

$$e_j = \langle \text{ COMMIT } : \ cmd(j) \ \rangle_r$$
$$e'_j = \langle \text{ COMMIT } : \ cmd(j) \ \rangle_{r'}$$
$$e'_k = \langle \text{ PREPARE } : \ cmd(k) \ \rangle$$

Suppose that event e_j occurs at replica r. Let $k > j$ be the minimal level for which there exist an event e'_k by some replica r', where $cmd(k)$ conflicts with $cmd(j)$. Then event e'_j may not occur.

Proof. By way of contradiction, assume the event e'_j exists. Lemma 1 implies that a level h exists, where $j < h \leq k$, in which there are already $2f + 1$ PREPARE-votes for $cmd(h)$ that conflicts with $cmd(j)$. This contradicts the minimality of k. □

Lemma 2. *Denote the following events:*

$$e_j = \langle \text{ COMMIT } : \ cmd(j) \ \rangle_r$$
$$e_h = \langle \text{ PROPOSE } , \ CC(cmd(h)) \ \rangle$$

Suppose that event e_j occurs at replica r. Let $h > j$ be the minimal level for which there exist an event e_h, where $cmd(h)$ conflicts with $cmd(j)$. Then $cmd(j)$ cannot ever become committed.

Proof. By Corollary 1, for every correct replica in $CC(cmd(h))$, there does not exist a COMMIT-vote event for $cmd(j)$. Since there are at least $f + 1$ correct replica in $CC(cmd(h))$, the lemma follows. □

Claim 1. *Denote the following events:*

$$e_j = \langle \text{ COMMIT } : \ cmd(j) \ \rangle_r$$
$$e_h = \langle \text{ PROPOSE } , \ CC(cmd(h)) \ \rangle$$

Let a command $cmd(h)$ (ever) become a committed decision. Then no conflicting command $cmd(j)$ at a lower level $j < h$ ever becomes committed.

Proof. By way of contradiction, assume $cmd(j)$ becomes committed. Then for some correct replica r, event e_j occurs. By Lemma 2, this implies that no event e_h carrying a COMMIT-certificate for a conflicting command $cmd(h)$ may occur. A fortiori, $cmd(h)$ may never become committed. □

6 Liveness

Providing liveness requires having infinitely many levels in which $2f+1$ correct replicas and a correct proposer are at the same level for sufficiently long to reach a decision. Algorithm 1 encapsulates the transition between levels via a beacon functionality, whose implementation is intentionally left unspecified, but needs to satisfy two requisites.

A beacon triggers entry into levels, and provides a proposal with the trigger. Replicas enter a level higher than their own upon receiving the beacon trigger, and vote for its proposal. Their votes determine a growing committed log-sequence.

We will say that a level maintains a *Lighthouse* property if the beacon collects $2f + 1$ votes for a proposal at the level. We will say that a level maintains a *Longest-Fork-Lives* (LFL) property if the beacon proposal for the level does not conflict with the highest-level COMMIT-certificate held by any correct replica. Liveness is guaranteed if infinitely often there are two consecutive levels that are both Lighthouse and LFL. This ensures that the first level of the pair obtains a COMMIT-certificate for a proposal, and the next level commits the proposal.

Liveness via Synchronized clocks

A realistic assumption made in production systems, e.g. Google Spanner [20], is that replicas have access to a globally synchronized clock with a known bounded skew. We can implement the beacon based on synchronized clocks by entering levels on pre-determined time slots at least 3Δ apart. We leave out the proof showing that after GST, this guarantees progress. One drawback of this approach is that there is no way to expedite the start of a new level, even if decisions advance rapidly.

Liveness using a Rotating Proposer

Another option is to explicitly build level-synchronization into the protocol via a proposer rotation. After GST, we can synchronize level entrance up to 2Δ using all-all broadcast. Naively, this brings us back to quadratic complexity. However, it is enough to synchronization once per rotation. This works as follows.

Normally, a replica enters the next level upon receiving a proposal carrying a COMMIT-certificate for the previous level. In this way, the algorithm moves at the network speed, and does not delay for some pre-determined synchronization points.

Transitions are handled differently at *rotation levels* $n, 2n, 3n, \ldots$. Upon preparing to enter a rotation level xn, a replica broadcasts to all replicas a synchronization message:

$$\langle \text{ENTER}, xn \rangle$$

Upon receiving $f + 1$ ENTER messages for level xn higher than its current level, a replica echoes with its own ENTER message. Upon receiving $2f + 1$ ENTER messages at a level xn higher than its own, a replica enters the level.

This scheme is essentially a Bracha "echo broadcast" [8]. After GST, is some correct replica obtains $2f + 1$ ENTER messages for xn at time T, then this echo-broadcast scheme guarantees that by time $T + 2\Delta$ time all correct replicas obtain $2f + 1$ ENTER messages for xn.

In order to guarantee progress for levels whose proposers are faulty, upon entering level-ℓ, a replica sets a timer for 4Δ. If the timer expires without receiving a proposal, a replica prepare to move to the next level. If this is a rotation level xn, the replica does not immediately enter it. As describes above, it sends an ENTER message for level xn, and synchronizes with others.

Algorithm 2 describes the beacon rotation-based implementation.

Algorithm 2 Beacon with a Rotating Proposer

(local variables and replica functionality are the same as Algorithm 1)

Beacon functionality at replica:

upon message $p = \langle \text{PROPOSE}, CC(cclevel), parent, newcmd(\ell) \rangle$
where either $(\ell = curlevel)$ or $(cclevel = curlevel)$:
beacon-proposal p;
pre-enter $(\ell + 1)$;

upon $2f + 1$ votes for for level-$(\ell - 1)$ proposal
pre-enter ℓ;

upon message $\langle \text{VOTE}, \ell, digest(c), digest(t) \rangle$:
(omitted for brevity: retrieve missing commands as needed, update $highCC$ and $highTail$)

upon pre-enter $\ell > curlevel$:
if $\ell \mod n = 0$ **then**
send $\langle \text{ENTER}, \ell \rangle$ to all replicas;
else
enter ℓ;

upon $f + 1$ messages $\langle \text{ENTER}, \ell \rangle$ ▷ Bracha echo
(omitted for brevity: retrieve commands up to h and process as needed)
pre-enter ℓ;

upon $2f + 1$ messages $\langle \text{ENTER}, \ell \rangle$ ▷ Bracha delivery
enter ℓ; ▷ enter rotation-level

upon $x \times 4\Delta$ elapsed since entering latest rotation-level $rlevel$:
pre-enter level $rlevel + x$;

upon enter ℓ :
$curlevel \Leftarrow \ell$;
if replica is proposer for $curlevel$ with input $newcmd$ **then**
send $\langle \text{PROPOSE}, highCC, cmd(curlevel - 1), newcmd(curlevel) \rangle$ to all replicas;

Liveness Proof of Algorithm 2

Lemma 3. *Let G be a time after GST. Let xn be any ratation-level for which the first entrance into level xn by any correct replica occurs after G. Denote this time by $T \geq G$. Then by $T + 2\Delta$ all correct replicas enter level xn.*

Proof. A correct replica enters a rotation level xn only after receiving $2f + 1$ ENTER messages for xn. Of these, $f + 1$ are from correct replicas, hence at the latest, they are received and echoed by all correct replicas by time $T + \Delta$. The echoes are received by all correct replicas by $T + 2\Delta$, and they enter xn. □

Lemma 4. *Let G be a time after GST. Let xn be any ratation-level for which the first entrance into level xn by any correct replica occurs after G. Let level ℓ, where $(x+1)n > \ell \geq xn$, have a correct proposer. Then no correct replica departs level ℓ by expiration.*

Proof. Assume to the contrary that some correct replica r departs level ℓ by expiration. We already know from Lemma 3 that all correct replicas enter xn within a 2Δ period. Hence, the level-ℓ proposer enters level ℓ at most 2Δ after replica r starts a 4Δ timer for the level. Since the proposal reaches r in Δ time, replica r's expiration could not happen. □

Lemma 5. *Let G be a time after GST. Let xn be any ratation-level for which the first entrance into level xn by any correct replica occurs after G. The levels ℓ, $\ell + 1$, $\ell + 2$, where $\ell \geq xn$, have correct proposers. Then the level-ℓ proposal becomes a committed decision.*

Proof. By Lemma 4, no correct replica departs level-ℓ by expiration. Therefore, the level-$(\ell + 1)$ proposer obtains responses either until is has a COMMIT-certificate for the level-ℓ proposal, or until it obtains responses by all correct replicas. Either way, after processing all response, the level-$(\ell + 1)$ proposal carries the highest COMMIT-certificate by any correct replica.

Similarly, by Lemma 4, no correct replica departs level-$(\ell + 1)$ by expiration. This time, the proposer for level-$(\ell + 2)$ is guaranteed to obtain $2f + 1$ votes for the level $(\ell + 1)$ proposal.

Consequently, the level-$(\ell + 2)$ proposal carries a COMMIT-certificate for the level $(\ell + 1)$ proposal, and it becomes committed with the votes for this level. □

Claim 2. *In every rotation, at least one proposal becomes a committed decision.*

Proof. Among n proposers, there is at least one stretch of three consecutive proposers. Accordig to Lemma 5, this guarantees for the third proposer's command to become committed. □

7 Wendy the View-Changing Finality Gadget

The Hot-Stuff abstract beacon functionality captures the requirements from a proposer mechanism to drive progress, similar to the Ω failure detector abstraction of Chandra and Toueg [16]. Different from a failure detector, a beacon may be realized by an auxiliary source. In particular, the idea is to harnesses recent advances in cryptocurrencies, specifically Proof-of-Work (PoW), to implement synchronization.

The auxiliary source must maintain the guarantee that infinitely often there are consecutive level pairs that are Lighthouse and LFL.

Lighthouse is satisfied through the scarcity and fixed-cadence of mining. These properties reflect core design principles for blockchains. They are guaranteed with high probability by proper parameter tuning of the PoW puzzles against available compute power. The timely arrival of proposals is generally assumed in PoW networks. It is further reinforced by considering for finalization only every (say) 100th block in the PoW blockchain.

In order to support Longest-Fork-Lives, PoW miners need to mine the longest-fork from highest COMMIT-certificate they know. Indeed, this would be the miners best strategy if they accept the decisions of a BFT engine as a *finalizing* the blockchain.

We refer to an instantiation of HS with an auxiliary PoW beacon satisfying the above requirement as Wendy.

Two previous approaches that utilize PoW as an auxiliary source of proposals is Solida [3] and Casper [12]. Solida uses PoW to drive progress in a permissionless settings, rotating a proposer into the system upon accepting a proposal. The Solida rotation mechanisms embraces the PBFT scheme, and could potentially be enhance with the HS solution framework.

Casper and Wendy

Buterin etal. introduce in [12] a finalizing engine called Casper, which greatly influenced Wendy's design.[2] In Casper, proposals are *blocks* of a public blockchain, and the BFT solution is a *finalizing* mechanism, motivated by Buterin in [11].

In Casper, beacons do not collect and push COMMIT-certificates. Instead, replicas broadcast their votes over a peer-to-peer network, which is assumed reliable and synchronous. Instead of proposals carrying a COMMIT-certificate, vote messages carry references to the the highest COMMIT-certificate known to the replica that sent them. Thus, a proposer in Wendy is *proactive* in collecting and brokering a COMMIT-certificate in order to promote its own interest. A proposer in Casper is *passive*, it only publishes a proposal, without actively participating in the voting.

In Casper's terms, a vote is cast over *an edge from an ancestor (source) to a proposal (target)*. The proposal becomes *justified* if the edge obtains $2f + 1$ votes, and the ancestor is justified. Additionally, the ancestor becomes *finalized* if the edge obtains $2f + 1$ votes, and the level-gap between the ancestor and the proposal is one. There is a direct mapping between these concepts and Wendy: An edge constitutes a VOTE, counting as a PREPARE-vote on the proposal. A justified proposal has $2f + 1$ PREPARE-votes, hence a COMMIT-certificate. When an edge has level-gap one, then it counts also as a COMMIT-vote on the ancestor. A finalized ancestor has $2f + 1$ COMMIT-votes, hence is becomes a committed-decision.

The HS algorithmic framework provides the following contributions over Casper. First, a proposer-driven framework reduces the communication complexity to $O(n)$ per level, and achieves overall optimal decision complexity. Second, it casts Casper in a conventional view-by-view, proposer driven framework. Last, it provides a rigorous proof of correctness.

8 Concluding Remarks

Hot-Stuff provides new foundations for BFT. It improves by $O(n^2)$ on the communication complexities of previous approaches, providing the first BFT protocol with linear complexity. These improvements can serve as a new basis for further optimizations and variations.

Accomplishing linearity does not result in a complex protocol, and on the contrary, simplifies it compared with previous solutions. This lends Hot-Stuff to robust implementations, and may be used for teaching.

Hot-Stuff is cast as a framework that separates liveness from safety. This fits modern BFT systems where Proof-of-Work provides progress guarantees, but not safety guarantees. In particular, Wendy is an instance of the Hot-Stuff framework that can serve as a finalizing engine for PoW chains.

Our goal in this manuscript has been foundational. More work is needed to fill in details concerning request batching, state checkpoint, detecting and removing faulty proposers, reconfiguration, and so on.

One question left open by this work is whether linear view-change is possible for BFT protocols with speculatively fast tracks. In all the known methods [26, 28, 25, 17, 5, 23], we can achieve a linear reduction, either by applying LVC, or by using threshold signature schemes (as demonstrated in [1]). However, combining the two to get linear-over-linear reduction is not obvious.

[2] Casper the Friendly Ghost, Hot Stuff the Little Devil, and Wendy the Good Little Witch, are fictional comic book characters from Harvey Comics.

References

[1] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. SBFT: Scaling up byzantine fault tolerance for blockchains. Under submission, 2018.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. ArXiv, https://arxiv.org/abs/1712.01367, 2017.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Sasha Spiegelman. Solida: A cryptocurrency based on reconfigurable byzantine consensus. In *OPODIS*, December, 2017.

[4] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Sec. Comput.*, 8(4):564–577, 2011.

[5] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[6] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[7] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, September 2004.

[8] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[9] Gene Zhang Bryan Fink, Eric Knauft. vsan: Modern distributed storage. *SIGOPS Oper. Syst. Rev.*, 51(1), 2017.

[10] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769, Thesis, 2016, University of Guelph.

[11] Vitalik Buterin. Minimal slashing conditions. https://medium.com/@VitalikButerin/minimal-slashing-conditions-20f0b500fc6c.

[12] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[13] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: Practical asynchronous byzantine agreement using cryptography (extended abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 123–132, New York, NY, USA, 2000. ACM.

[14] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[16] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[17] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[18] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[19] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[21] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[23] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 119–128, New York, NY, USA, 2007. ACM.

[24] F.P. Junqueira, B.C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.

[25] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, January 2010.

[26] K. Kursawe. Optimistic byzantine agreement. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 262 – 267, 2002.

[27] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[28] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215, July 2006.

[29] Santoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https:// bitcoin.org/bitcoin.pdf, December 2008.

[30] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[31] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[32] Michael K. Reiter. The rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, UK, 1995. Springer-Verlag.

[33] Victor Shoup. Practical threshold signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag.