HotStuff: BFT Consensus in the Lens of Blockchain

Stefanos Chaliasos

HotStuff: BFT Consensus in the Lens of Blockchain

Maofan Yin[1,2], Dahlia Malkhi[2], Michael K. Reiter[2,3], Guy Golan Gueta[2], and Ittai Abraham[2]

[1]Cornell University, [2]VMware Research, [3]UNC-Chapel Hill

Abstract

We present HotStuff, a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called *responsiveness*—and with communication complexity that is linear in the number of replicas. To our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting these combined properties. HotStuff is built around a novel framework that forms a bridge between classical BFT foundations and blockchains. It allows the expression of other known protocols (DLS, PBFT, Tendermint, Casper), and ours, in a common framework.

Our deployment of HotStuff over a network with over 100 replicas achieves throughput and latency comparable to that of BFT-SMaRt, while enjoying linear communication footprint during leader failover (vs. cubic with BFT-SMaRt).

1 Introduction

Byzantine fault tolerance (BFT) refers to the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of its components while taking actions critical to the system's operation. In the context of state machine replication (SMR) [35, 47], the system as a whole provides a replicated service whose state is mirrored across n deterministic replicas. A BFT SMR protocol is used to ensure that non-faulty replicas agree on an order of execution for client-initiated service commands, despite the efforts of f Byzantine replicas. This, in turn, ensures that the $n-f$ non-faulty replicas will run commands identically and so produce the same response for each command. As is common, we are concerned here with the partially synchronous communication model [25], whereby a known bound Δ on message transmission holds after some unknown *global stabilization time* (GST). In this model, $n \geq 3f+1$ is required for non-faulty replicas to agree on the same commands in the same order (e.g., [12]) and progress can be ensured deterministically only after GST [27].

When BFT SMR protocols were originally conceived, a typical target system size was $n = 4$ or $n = 7$, deployed on a local-area network. However, the renewed interest in Byzantine fault-tolerance brought about by its application to blockchains now demands solutions that can scale to much larger n. In contrast to *permissionless* blockchains such as the one that supports Bitcoin, for example, so-called *permissioned* blockchains involve a fixed set of replicas that collectively maintain an ordered ledger of commands or, in other words, that support SMR. Despite their permissioned nature, numbers of replicas in the hundreds or even thousands are envisioned (e.g., [42, 30]). Additionally, their deployment to wide-area networks requires setting Δ to accommodate higher variability in communication delays.

What is HotStuff

Byzantine fault-tolerant (BFT) State machine replication (SMR) protocol for the partially synchronous model.

What is HotStuff

Byzantine fault-tolerant (BFT) State machine replication (SMR) protocol for the partially synchronous model.

- **Byzantine Fault Tolerance (BFT):** ability of a computing system to endure arbitrary failures of its component while taking actions critical to the system's operation.

What is HotStuff

Byzantine fault-tolerant (BFT) State machine replication (SMR) protocol for the partially synchronous model.

- **Byzantine Fault Tolerance (BFT):** ability of a computing system to endure arbitrary failures of its component while taking actions critical to the system's operation.
- **State Machine Replication (SMR):** the system as a whole provides a replicated service whose state is mirrored across n deterministic replicas.

What is HotStuff

Byzantine fault-tolerant (BFT) State machine replication (SMR) protocol for the partially synchronous model.

- **Byzantine Fault Tolerance (BFT):** ability of a computing system to endure arbitrary failures of its component while taking actions critical to the system's operation.
- **State Machine Replication (SMR):** the system as a whole provides a replicated service whose state is mirrored across n deterministic replicas.
- Ensure that non-faulty replicas agree on an order of execution for client-initiated service commands, despite the efforts of f Byzantine replicas.

Synchrony – Asynchrony

- **Asynchrony:** adversary can delay messages by any finite amount.

Synchrony – Asynchrony

- **Asynchrony:** adversary can delay messages by any finite amount.
- **Synchrony:** adversary can delay messages by some known Δ

Synchrony – Asynchrony

- **Asynchrony:** adversary can delay messages by any finite amount.
- **Synchrony:** adversary can delay messages by some known Δ
- **Partial Synchrony:**

Synchrony – Asynchrony

- **Asynchrony:** adversary can delay messages by any finite amount.
- **Synchrony:** adversary can delay messages by some known Δ
- **Partial Synchrony:**
  - adversary can delay messages by any finite amount

Synchrony – Asynchrony

- **Asynchrony:** adversary can delay messages by any finite amount.
- **Synchrony:** adversary can delay messages by some known Δ
- **Partial Synchrony:**
  - adversary can delay messages by any finite amount
  - until some unknown finite point in time called GST (Global Stabilization Time)

Synchrony – Asynchrony

- **Asynchrony:** adversary can delay messages by any finite amount.
- **Synchrony:** adversary can delay messages by some known Δ
- **Partial Synchrony:**
  - adversary can delay messages by any finite amount
  - until some unknown finite point in time called GST (Global Stabilization Time)
  - adversary can delay messages by some known Δ

Safety – Liveness

- **Liveness:** property is a guarantee that each component will eventually decide on a value (this can be referred to as termination).
- **Safety:** property is a guarantee that different components will never decide on different values (this can be referred to as agreement).

Scaling Consensus

Nakamoto Consensus	PBFT
Minimised setup (Pow)	PKI setup
Worldwide, large scale	LAN, 4 or 7
Finality in Minutes	Sub-second finality
7-15 TPS	1000s TPS
Linear Communication	Quadratic Communication
Liveness against adaptive (CR)	Weak liveness against DOS
Safety depends in synchrony	Always Safe (partial synchrony)
Large energy consumption	Efficient energy consumption

Scaling Consensus

Nakamoto Consensus	HotStuff BFT SMR	PBFT
Minimised setup (Pow)	PoW, PoS or **PKI**	PKI setup
Worldwide, large scale	10s 100s 1000s	LAN, 4 or 7
Finality in Minutes	Sub-second finality	Sub-second finality
7-15 TPS	1000s TPS	1000s TPS
Linear Communication	Linear Communication	Quadratic Communication
Liveness against adaptive (CR)	Always Live (partial synchrony)	Weak liveness against DOS
Safety depends in synchrony	Always Safe (partial synchrony)	Always Safe (partial synchrony)
Large energy consumption	Efficient energy consumption	Efficient energy consumption

PBFT (Scaling challenge)

- A stable leader can drive a consensus decision in **two** phases of message exchanges.
  - First phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n−f) votes.
  - The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.

PBFT (Scaling challenge)

- A stable leader can drive a consensus decision in **two** phases of message exchanges.
    - First phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n–f) votes.
    - The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.
- The algorithm for a new leader to collect information and propose it to replicas, called a **view-change** is the epicenter of replication.

PBFT (Scaling challenge)

- A stable leader can drive a consensus decision in **two** phases of message exchanges.
  - First phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n–f) votes.
  - The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.
- The algorithm for a new leader to collect information and propose it to replicas, called a **view-change** is the epicenter of replication.
- View-change based on the two-phase paradigm is *complicated, bug-prone, not scalable.*

PBFT (Scaling challenge)

- A stable leader can drive a consensus decision in **two** phases of message exchanges.
  - First phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n–f) votes.
  - The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.
- The algorithm for a new leader to collect information and propose it to replicas, called a **view-change** is the epicenter of replication.
- View-change based on the two-phase paradigm is *complicated, bug-prone, not scalable.*
- Zyzzyva SOSP 07 Best paper award (Bug found in 2017)

PBFT (Scaling challenge)

- A stable leader can drive a consensus decision in **two** phases of message exchanges.
  - First phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of (n–f) votes.
  - The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.
- The algorithm for a new leader to collect information and propose it to replicas, called a **view-change** is the epicenter of replication.
- View-change based on the two-phase paradigm is *complicated, bug-prone, not scalable.*
- Zyzzyva SOSP 07 Best paper award (Bug found in 2017)
- HotStuff propose a **three phase** protocol.

HotStuff Contributions

- **Linear View Change**
- **Optimistic Responsiveness:** After GST, any correct leader, once designated, needs to wait just for the first $n - f$ responses to guarantee that it can create a proposal that will make progress.

Asynchronous BFT protocols (after GST)

Protocol	Authenticator complexity: Correct leader	Leader failure (view-change)	f leader failures	Responsiveness
DLS [25]	$O(n^4)$	$O(n^4)$	$O(n^4)$	
PBFT [20]	$O(n^2)$	$O(n^3)$	$O(fn^3)$	✓
SBFT [30]	$O(n)$	$O(n^2)$	$O(fn^2)$	✓
Tendermint [15] / Casper [17]	$O(n^2)$	$O(n^2)$	$O(fn^2)$	
Tendermint* / Casper*	$O(n)$	$O(n)$	$O(fn)$	
HotStuff	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(fn)}$	✓

*Signatures can be combined using threshold signatures, though this optimization is not mentioned in their original works.

- *Authenticator complexity* is the sum, over all replicas, of the number of authenticators received by replica i in the protocol to reach a consensus decision after GST. An *authenticator* is either a partial signature or a signature.

HotStuff Model

- n fixed replicas; Adversary controls f replicas, $\mathbf{n = 3f + 1}$.
- Byzantine faulty replicas coordinated by an adversary that learns everything.
- Each view v has a single primary.
- **Network communication** is point-to-point, authenticated and reliable
  - one correct replica receives a message from another correct replica if and only if the later sent that message
  - broadcast: if correct replica sending the same point-to-point messages to all replicas, including itself
- A node can have multiple roles.
- Partial synchrony model.
- Protocol guarantees progress if the system remains stable (messages arrive within Δ time).
- Protocol guarantees safety always.

Basic HotStuff (1)

- Solves the SMR problem
- Deciding on a growing log of command requests by clients
- A client sent a command to all replicas, and waits for response from (f + 1) of them.
- The protocol works in a succession of views accompanied by an incrising number.
- Each *viewNumber* has a unique dedicated leader known to all.
- Quorum Certificate (**QC**) is a collection of (n - f) votes over a leader proposal, assosiated with a particular node and a view number.

Basic HotStuff (2)

- Protocol is described as an iterated view-by-view loop.
- Three phase protocol
- For simplicity, in the begin of each view **v** a primary broadcasts (sends) a value **x** (high QC from replicas, client's commands)

Force Primary to say just one thing: Safety (1)

- Primary <sends x, v> to all replicas

Force Primary to say just one thing: Safety (1)

- Primary <sends x, v> to all replicas
- Each replica responds <ack x, v> on the first message in view *v* from the primary

Force Primary to say just one thing: Safety (1)

- Primary <sends x, v> to all replicas
- Each replica responds <ack x, v> on the first message in view *v* from the primary
- Primary waits for (n - f) <ack x, v> and sends <lock x, v> containing (n - f) ack signatures

Force Primary to say just one thing: Safety (1)

- Primary <sends x, v> to all replicas
- Each replica responds <ack x, v> on the first message in view *v* from the primary
- Primary waits for (n - f) <ack x, v> and sends <lock x, v> containing (n - f) ack signatures
- Replica accepts <lock x, v> if there are (n -f) valid ack signatures

Force Primary to say just one thing: Safety (1)

- Primary <sends x, v> to all replicas
- Each replica responds <ack x, v> on the first message in view *v* from the primary
- Primary waits for (n - f) <ack x, v> and sends <lock x, v> containing (n - f) ack signatures
- Replica accepts <lock x, v> if there are (n -f) valid ack signatures
- *Claim:* if honest replica locks on <x, v>, no honest replica locks on <x', v>

Force Primary to say just one thing: Safety (1)

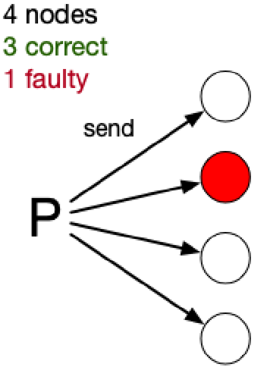
- Primary <sends x, v> to all replicas
- Each replica responds <ack x, v> on the first message in view *v* from the primary
- Primary waits for (n - f) <ack x, v> and sends <lock x, v> containing (n - f) ack signatures
- Replica accepts <lock x, v> if there are (n -f) valid ack signatures
- *Claim:* if honest replica locks on <x, v>, no honest replica locks on <x', v>
- *Proof:* there are at least (n - 2f) honest for <ack x,v>, they block any other (n - f > 2f) set

Force Primary to say just one thing: Safety (1)

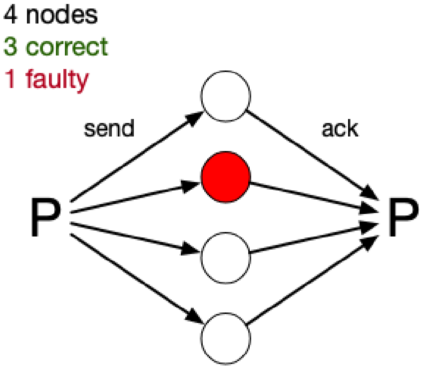
4 nodes
3 correct
1 faulty

P

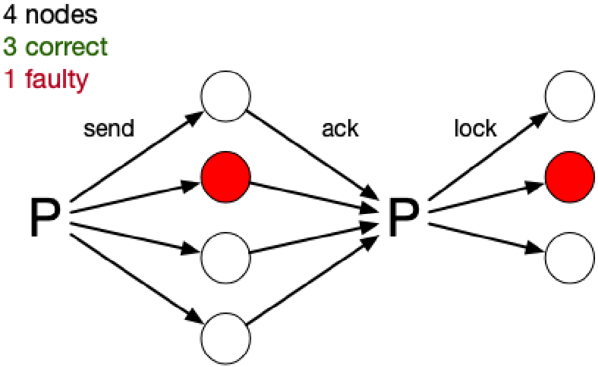
Force Primary to say just one thing: Safety (1)



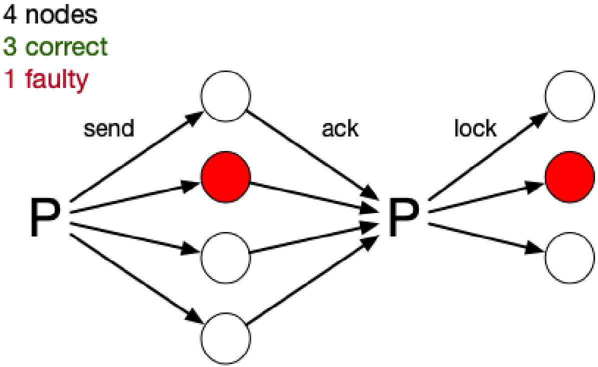
Force Primary to say just one thing: Safety (1)



Force Primary to say just one thing: Safety (1)



Force Primary to say just one thing: Safety (1)



How to never forget a commitment (1)

Suppose malicious primary causes just one replica to commit to x
Must guarantee that newer primaries will never contradict and cause a commit to x'

- Primary sends <lock x, v> to all replicas

How to never forget a commitment (1)

Suppose malicious primary causes just one replica to commit to x
Must guarantee that newer primaries will never contradict and cause a commit to x'

- Primary sends <lock x, v> to all replicas
- Each replica responds <lock ack x,v> on the first message in view *v* from the primary

How to never forget a commitment (1)

Suppose malicious primary causes just one replica to commit to x
Must guarantee that newer primaries will never contradict and cause a commit to x'

- Primary sends <lock x, v> to all replicas
- Each replica responds <lock ack x,v> on the first message in view *v* from the primary
- Primary waits for (n - f) <lock ack x,v> and sends <commit x,v> containing (n - f) lock signatures

How to never forget a commitment (1)

Suppose malicious primary causes just one replica to commit to x
Must guarantee that newer primaries will never contradict and cause a commit to x'

- Primary sends <lock x, v> to all replicas
- Each replica responds <lock ack x,v> on the first message in view *v* from the primary
- Primary waits for (n - f) <lock ack x,v> and sends <commit x,v> containing (n - f) lock signatures
- Replica commits to <x,v> if there are (n - f) valid lock signatures

How to never forget a commitment (1)

Suppose malicious primary causes just one replica to commit to x
Must guarantee that newer primaries will never contradict and cause a commit to x'

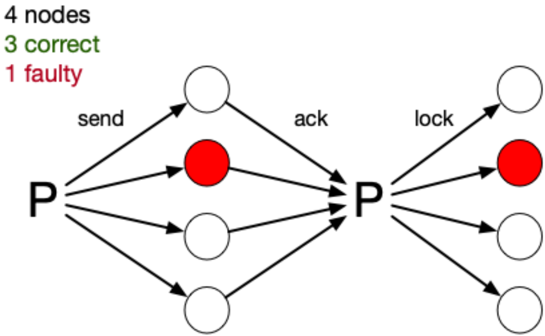
- Primary sends <lock x, v> to all replicas
- Each replica responds <lock ack x,v> on the first message in view *v* from the primary
- Primary waits for (n - f) <lock ack x,v> and sends <commit x,v> containing (n - f) lock signatures
- Replica commits to <x,v> if there are (n - f) valid lock signatures
- *Commit rule:* guarentees that there is enough people locked in the value of a commit

How to never forget a commitment (1)

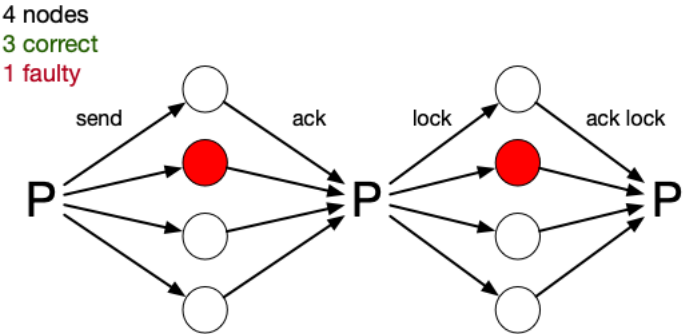
Suppose malicious primary causes just one replica to commit to x
Must guarantee that newer primaries will never contradict and cause a commit to x'

- Primary sends <lock x, v> to all replicas
- Each replica responds <lock ack x,v> on the first message in view *v* from the primary
- Primary waits for (n - f) <lock ack x,v> and sends <commit x,v> containing (n - f) lock signatures
- Replica commits to <x,v> if there are (n - f) valid lock signatures
- *Commit rule:* guarentees that there is enough people locked in the value of a commit
- Safe leader replacement

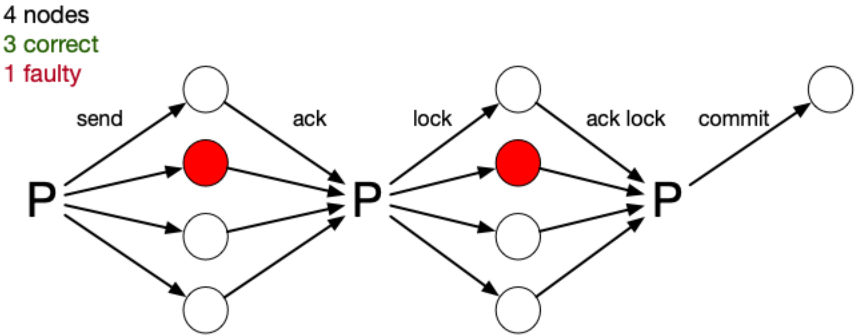
How to never forget a commitment (2)



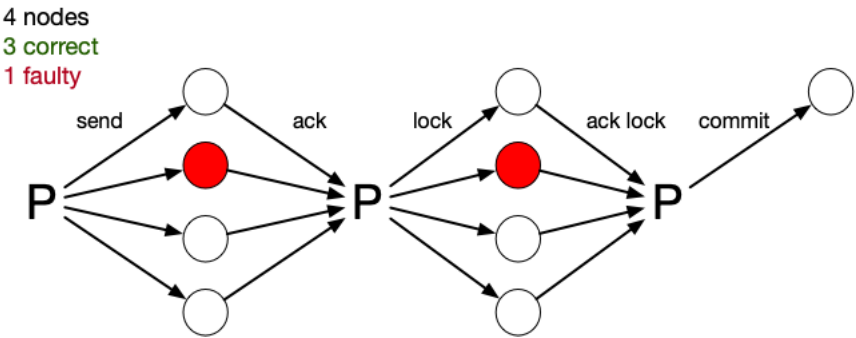
How to never forget a commitment (2)



How to never forget a commitment (2)



How to never forget a commitment (2)



- When a new Primary comes, it will check if other values are locked, and it will verify that the commited value is the same with the locked values from (n - f) nodes.

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x’,v’>
  - Lock = maximal view where replica saw a <lock x’,v’> with (n - f) <ack x’,v’> signatures
  - How can the primary of view v > v’ convice a locked replica that <x’,v’> was not committed?

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures
  - How can the primary of view v > v' convice a locked replica that <x',v'> was not committed?
  - For a locked replica on <x',v'> to change its mind, Primary must:

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures
  - How can the primary of view v > v' convice a locked replica that <x',v'> was not committed?
  - For a locked replica on <x',v'> to change its mind, Primary must:
    - Show a lock of a higher view v" > v'

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x’,v’>
  - Lock = maximal view where replica saw a <lock x’,v’> with (n - f) <ack x’,v’> signatures
  - How can the primary of view v > v’ convice a locked replica that <x’,v’> was not committed?
  - For a locked replica on <x’,v’> to change its mind, Primary must:
    - Show a lock of a higher view v” > v’
    - Use the same value x’

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures
  - How can the primary of view v > v' convice a locked replica that <x',v'> was not committed?
  - For a locked replica on <x',v'> to change its mind, Primary must:
    - Show a lock of a higher view v" > v'
    - Use the same value x'
    - Otherwite the replica ignores Primary

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x’,v’>
  - Lock = maximal view where replica saw a <lock x’,v’> with (n - f) <ack x’,v’> signatures
  - How can the primary of view v > v’ convice a locked replica that <x’,v’> was not committed?
  - For a locked replica on <x’,v’> to change its mind, Primary must:
    - Show a lock of a higher view v” > v’
    - Use the same value x’
    - Otherwite the replica ignores Primary
- New Primary can:

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x’,v’>
  - Lock = maximal view where replica saw a <lock x’,v’> with (n - f) <ack x’,v’> signatures
  - How can the primary of view v > v’ convice a locked replica that <x’,v’> was not committed?
  - For a locked replica on <x’,v’> to change its mind, Primary must:
    - Show a lock of a higher view v” > v’
    - Use the same value x’
    - Otherwite the replica ignores Primary
- New Primary can:
  - <send x,v>

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures
  - How can the primary of view v > v' convice a locked replica that <x',v'> was not committed?
  - For a locked replica on <x',v'> to change its mind, Primary must:
    - Show a lock of a higher view v" > v'
    - Use the same value x'
    - Otherwite the replica ignores Primary
- New Primary can:
  - <send x,v>
    - Will be accepted only if no replica is locked

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures
  - How can the primary of view v > v' convice a locked replica that <x',v'> was not committed?
  - For a locked replica on <x',v'> to change its mind, Primary must:
    - Show a lock of a higher view v" > v'
    - Use the same value x'
    - Otherwite the replica ignores Primary
- New Primary can:
  - <send x,v>
    - Will be accepted only if no replica is locked
  - <send y,v, lock y,w> where <lock y,w> is a valid lock (contains n - f ack signatures)

How to replace a primary: Safety

If a replica is locked, it does not easily change its mind

- Replica may be locked on <x',v'>
  - Lock = maximal view where replica saw a <lock x',v'> with (n - f) <ack x',v'> signatures
  - How can the primary of view v > v' convice a locked replica that <x',v'> was not committed?
  - For a locked replica on <x',v'> to change its mind, Primary must:
    - Show a lock of a higher view v" > v'
    - Use the same value x'
    - Otherwite the replica ignores Primary
- New Primary can:
  - <send x,v>
    - Will be accepted only if no replica is locked
  - <send y,v, lock y,w> where <lock y,w> is a valid lock (contains n - f ack signatures)
    - Will be accepted by any replica only if it is locked on value y, or locked on a view v' < w

How to replace a primary: Liveness (1)

- Each replica sends their lock to the primary

How to replace a primary: Liveness (1)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock

How to replace a primary: Liveness (1)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty

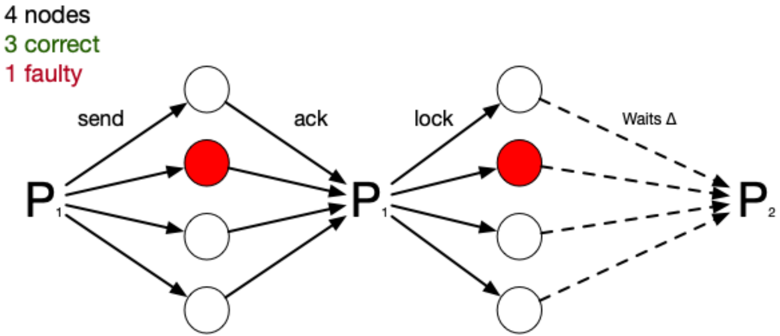
How to replace a primary: Liveness (1)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty
- In theory, in the partial synchrony model: requires to wait Δ

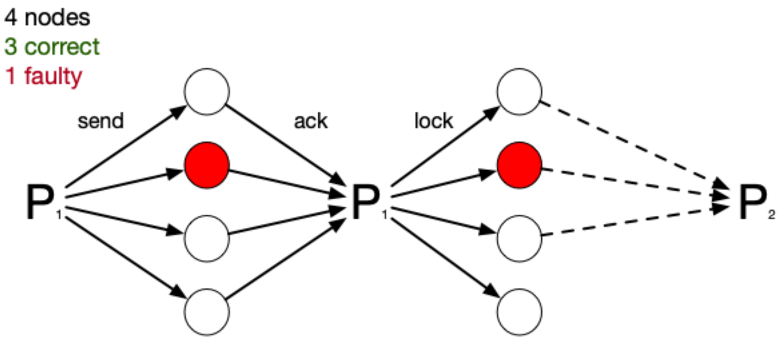
How to replace a primary: Liveness (1)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty
- In theory, in the partial synchrony model: requires to wait Δ
  - Liveness attack requires a malicious and sophisticated attacker

How to replace a primary: Liveness (2)



How to replace a primary: Liveness (3) – Hidden lock



How to replace a primary: Liveness (4)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty
- In theory, in the partial synchrony model: requires to wait Δ
  - Liveness attack requires a malicious and sophisticated attacker
- HotStuff removes this requirement by using one more round

How to replace a primary: Liveness (4)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty
- In theory, in the partial synchrony model: requires to wait Δ
  - Liveness attack requires a malicious and sophisticated attacker

- HotStuff removes this requirement by using one more round
  - Removes the 'hidden lock' problem, by making sure a key = 'lock-precursor' is always visible

How to replace a primary: Liveness (4)

- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty
- In theory, in the partial synchrony model: requires to wait Δ
  - Liveness attack requires a malicious and sophisticated attacker

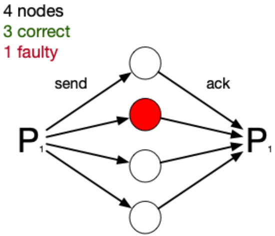
- HotStuff removes this requirement by using one more round
  - Removes the 'hidden lock' problem, by making sure a key = 'lock-precursor' is always visible
  - Important when the gap between Δ and the actual delay $\delta \ll \Delta$ is large

How to replace a primary: Liveness (4)

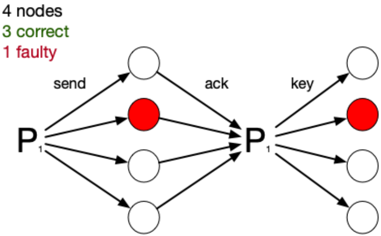
- Each replica sends their lock to the primary
- Primary uses the lock of the highest view, or <send x,v> if there is no lock
- Live if you hear from all non-faulty
- In theory, in the partial synchrony model: requires to wait Δ
  - Liveness attack requires a malicious and sophisticated attacker

- HotStuff removes this requirement by using one more round
  - Removes the 'hidden lock' problem, by making sure a key = 'lock-precursor' is always visible
  - Important when the gap between Δ and the actual delay $\delta \ll \Delta$ is large
  - **Optimistic responsiveness:** protocol makes progress at the speed of $\delta \ll \Delta$ when Primaries are honest

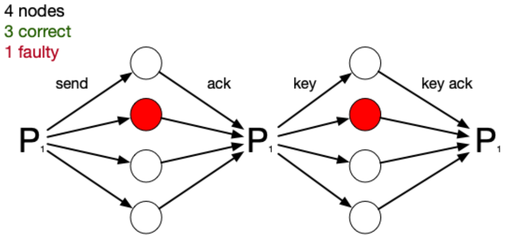
Force Primary to say just one thing: Safety (1)



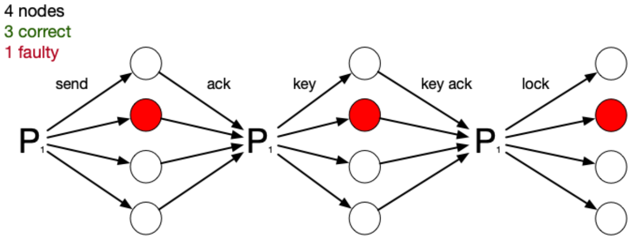
Force Primary to say just one thing: Safety (1)



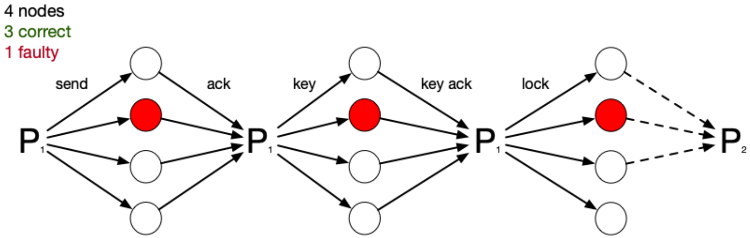
Force Primary to say just one thing: Safety (1)



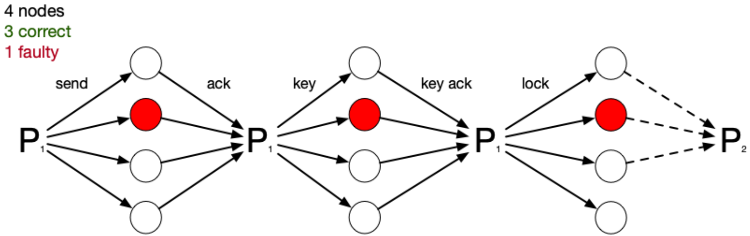
Force Primary to say just one thing: Safety (1)



Force Primary to say just one thing: Safety (1)

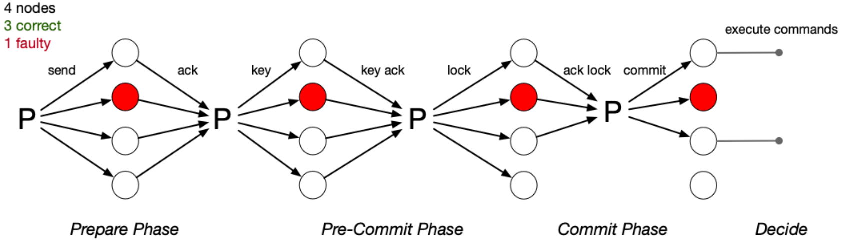


Force Primary to say just one thing: Safety (1)



- Guarantee that Primary 2 will receive a key from honest nodes to convice the ‘hidden lock’ node.

Complete Protocol



Chained HotStuff

- Improve Basic HotStuff protocol utility while at the same time considerable simplifying it.
- Basic idea: change view on every PREPARE phase, so each proposal has its own view.
- Reduces the number of messages and allows for pipelining of decisions.

Implementation

- ~4000 LOC C++.
- Simplify the code by extracting liveness mechanism from the HotStuff mechanism into a module named *Pacemaker*.
- Pacemaker is a mechanism that guarantees progress after GST.

Evaluation

- Compare HotStuff with state-of-the art (BFT-SMaRt)
- Amazon EC2 16 CPU per instance (one replica per instance)

Base Performance

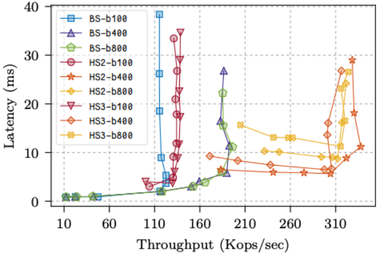


Figure 4: Throughput vs. latency with different choices of batch size, 4 replicas, 0/0 payload.

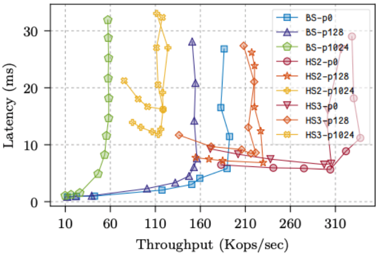


Figure 5: Throughput vs. latency with different choices of payload size, 4 replicas, batch size of 400.

- HotStuff achieves comparable latency performance to BFT-SMaRt, and its maximum throughput outperformes BFT-SMaRt.

Scalability (1)

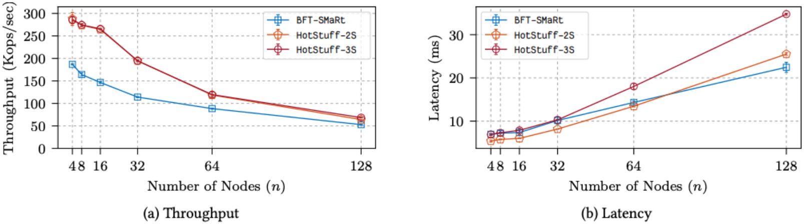


Figure 6: Scalability with 0/0 payload, batch size of 400.

- Better throughput than BFT-SMaRt, while latency still comparible.

Scalability (2)

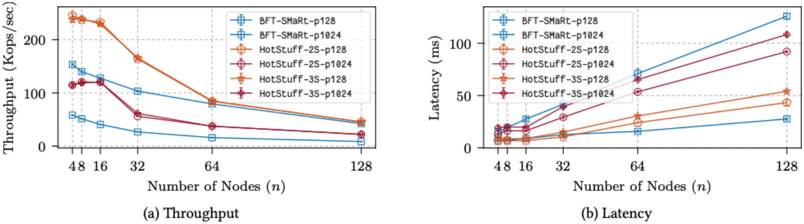


Figure 7: Scalability for 128/128 payload or 1024/1024 payload, with batch size of 400.

- Due to its quadratic bandwidth cost, the throughput of BFT-SMaRt scales worse than HotStuff for reasonably large (1024-byte) payload size.

View Change

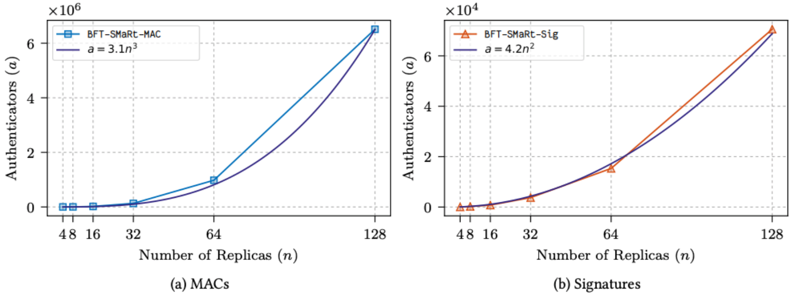


Figure 10: Number of extra authenticators used for each BFT-SMaRt view change.

- HotStuff has no ‘extra’ cost for leader changes by definition.