Byzantine State Machine Replication in the age of Blockchains: *fundamentals and recent results*

Alysson Bessani
Our (Ambitious) Agenda for Today

- Protocols
  - PBFT
  - Nakamoto consensus
  - Hybrid consensus
  - Chained consensus:
    - Streamlet
    - Gasper (Ethereum 2)
    - HotStuff

- Recent results
  - Order fairness
  - Scalability: Kauri and scalable communication
  - Geo-distribution

- Beyond consensus
  - Diversity
  - Confidentiality
Practical Byzantine Fault Tolerance (BFT)

a.k.a Byzantine/BFT active/State Machine replication
PBFT – Practical Byzantine Fault Tolerance
(Castro and Liskov, 1999, 2002)

• The “Byzantine version of VSR/Paxos”
• Primary-based BFT SMR algorithm
  • System evolves in views, numbered sequentially
  • In each view $v$, one server is the primary (leader), the others are the backups
    • “Primary of view $v” = v \text{ mod } N$
• Efficient and fast
  • Uses message authentication codes (MACs), instead of asymmetric crypto signatures
    (this is not a very important feature today, so we’ll ignore it)
  • Optimal in terms of latency (3 comm. steps) and resilience ($3f+1$)
• Prevention-tolerance mix
  • Clients’ requests and server messages are signed
  • Clients and servers discard messages with invalid signatures
PBFT System Model

• Asynchronous distributed system (see next slide)
• Network can lose, delay, reorder and duplicate messages; but cannot do that indefinitely
  • i.e., they require fair links to implement reliable channels
• Byzantine fault model
  • with fault independence (i.e., no common mode faults)
  • requires 3f+1 replicas to tolerate f faults
• Cryptography
  • PK signatures to facilitate the protocol presentation
  • Cryptographic hashes
• Adversary cannot break cryptographic primitives
PBFT Service Properties

- **Deterministic** replicated service

- **Service’ safety:**
  - The replicated service should behave as its centralized counterpart (**Linearizability**)
  - Even with malicious replicas compromising their states

- **Service’ liveness** (requires synchrony):
  - A command issued by a correct client will eventually be executed (**Wait-freedom**)
    
    *if the network transmission delay doesn’t grow faster than real time*
  - This is satisfied by the **partial synchronous** model: the system is asynchronous, but there is an unknown global stabilization time after which it become synchronous
PBFT Algorithm (2002 version)

• Algorithm essentials:
  • Two operation modes: normal operation and view change
  • A checkpoint protocol is executed periodically to truncate logs
  • A state transfer protocol is executed when needed (after a replica recovery)

• Algorithm outline:
  • All messages are signed
  • Clients multicast a request with a command and a timestamp to all servers
  • Servers reach agreement on the request to be delivered w/ a sequence number
  • Client waits for f+1 matching replies (at least one from a correct server)
PFT Consensus Protocol (Single-shot PBFT)

There are $n$ processes that want to solve consensus while tolerating $f < \frac{n}{3}$ Byzantine faults.

Byzantine Consensus property:

- **Agreement**: no two correct processes decide on different values.
- **Termination**: every correct process eventually decides on a value.
- **Validity**: $v$ must be the decided value if all correct processes propose the same value $v$.

According to an application-dependent predicate that can be verified locally in polynomial time (Bravo et al., 2022)
Single-shot PBFT: Separating Safety from Liveness

- The protocol works in a sequence of views.
- Each view has one leader, and processes use $\text{leader}(v) = p((v-1) \mod n)+1$ to determine the leader — The leader is the only one capable of proposing values in its view.
- A Synchronizer is responsible for changing views.
  - It must ensure enough processes stay in the same view for enough time
  - Formally, this must be ensured after GST in a partially synchronous system model

---

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>$\ldots$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>$\ldots$</td>
<td>$i$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>$\ldots$</td>
<td>$i$</td>
</tr>
</tbody>
</table>

Enough time for a $j+1$ quorum of correct processes to decide a proposed value!
Single-shot PBFT: The phases of a view

1. **Pre-processing**
   Processes inform the leader about the latest value it *prepared* in a previous view

2. **Propose**
   Leader computes a *valid* proposal and sends it along with some supporting information to all others

3. **Prepare**
   Processes communicate with each other to avoid to be fooled by a Byzantine leader

4. **Commit**
   Processes communicate with each other to commit a value

![Diagram showing the phases of a view](image)
Single-shot PBFT Protocol – part I

1. **upon** `new_view(v)`
2. \[ \text{curr\_view} \leftarrow v; \]
3. \[ \text{voted} \leftarrow \text{FALSE}; \quad \text{shows that } p_i \text{ has not yet received any proposal from the leader of curr\_view} \]
4. \[ \text{send} \langle \text{NEWLEADER}(\text{curr\_view}, \text{prepared\_view}, \text{prepared\_val}, \text{cert}) \rangle^i \text{ to leader(} \text{curr\_view}) \rangle; \]
Single-shot PBFT Protocol – part II

5 when received \{\text{NEWLEADER}(v, \text{view}_j, \text{val}_j, \text{cert}_j)\}_{j \mid p_j \in Q} = M \text{ for a quorum } Q

6 \textbf{pre: curr\_view} = v \land p_i = \text{leader}(v) \land (\forall m \in M. \text{ValidNewLeader}(m));

7 \text{if } \exists j. \text{view}_j = \max\{\text{view}_k \mid p_k \in Q\} \neq 0 \text{ then}

8 \quad \text{send } \langle \text{PROPOSE}(v, \text{val}_j, M) \rangle_i \text{ to all;}

9 \text{else}

10 \quad \text{send } \langle \text{PROPOSE}(v, \text{my\_val}(), M) \rangle_i \text{ to all;}

Pre-condition for proposing:
- Process is the leader for its current view $v$
- Received valid NEWLEADER for $v$ from a quorum $(2f+1$ processes)
Single-shot PBFT Protocol – part III

11 when received \(\langle\text{PROPOSE}(v, x, \_\rangle)\rangle_j = m\)
12 \[\text{pre: curr\_view = } v \land \text{voted = FALSE} \land \text{SafeProposal}(m);\]
13 curr\_val \leftarrow x;
14 voted \leftarrow \text{TRUE};
15 send \(\langle\text{PREPARED}(v, \text{hash(curr\_val)\rangle)}\rangle_i \text{ to all};\)

Using \(M\) from the PROPOSE message, each process can check whether the proposed value is valid.
Single-shot PBFT Protocol – part IV

16 when received \{⟨\text{PREPARED}(v, h)⟩_j \mid p_j \in Q\} = C for a quorum \( Q \)

\[ p! \]

in this phase, processes can identify whether the leader sent different proposals or not

\[ p" \]

\[ p# \]

\[ p$ \]

17 \hspace{1cm} \text{pre: curr\_view} = v \land \text{voted} = \text{TRUE} \land \text{hash(curr\_val)} = h;

\[ ⟨\text{NEWLEADER…}⟩ \]

18 \hspace{1cm} \text{prepared\_val} \leftarrow \text{curr\_val};

\[ ⟨\text{PROPOSE}(1, X, M)⟩_1 \]

19 \hspace{1cm} \text{prepared\_view} \leftarrow \text{curr\_view};

\[ ⟨\text{PROPOSE}(1, Y, M)⟩_1 \]

20 \hspace{1cm} \text{cert} \leftarrow C;

\[ ⟨\text{PROPOSE}(1, X, M)⟩_1 \]

21 \hspace{1cm} \text{send} \langle\text{COMMITTED}(v, h)\rangle_i \text{ to all;}

\[ ⟨\text{PREPARED}(X,…)⟩_1 \]

\[ ⟨\text{PREPARED}(X,…)⟩_1 \]

\[ ⟨\text{PREPARED}(X,…)⟩_1 \]

Is it possible to have two correct processes “preparing” different values (\( \text{val} \) and \( \text{val}' \)) in the same view \( v \)?
Single-shot PBFT Protocol – part IV

16 when received \(\{\{\text{PREPARED}(v, h)\}_j \mid p_j \in Q\} = C\) for a quorum \(Q\)

\[\begin{align*}
\text{pre:} & \quad \text{curr\_view} = v \land \text{voted} = \text{TRUE} \land \text{hash(curr\_val)} = h; \\
\text{prepared\_val} & \leftarrow \text{curr\_val}; \\
\text{prepared\_view} & \leftarrow \text{curr\_view}; \\
\text{cert} & \leftarrow C; \\
\text{send} & \{\text{COMMITTED}(v, h)\}_i \text{ to all};
\end{align*}\]

\(\text{in this phase, processes can identify whether the leader sent different proposals or not}\)

\(\text{at this point, we say that the process prepared the value and creates a prepared certificate}\)

Assume that the leader is correct. Can a process decide (terminate) at this point?
Single-shot PBFT Protocol – part V

22 when received \{\text{COMMITTED}(v, h)\}_{j} \mid p_j \in Q \}
   for a quorum Q
\begin{align*}
\text{pre: } & \text{curr\_view} = \text{prepared\_view} = v \land \\
& \text{hash(curr\_val)} = h;
\end{align*}
24 decide(curr\_val);  

With one additional phase, each process knows other processes will decide the same value it decided

Valid proposal:
- If a correct process decides $X$, then only $X$ is a valid proposal for the next views
- If at least $f + 1$ correct processes prepare $X$, then only $X$ is a valid proposal for the next views
- Otherwise, mayval() (which contains the process proposal) is a valid proposal
BFT-SMaRt

• Byzantine/Crash fault tolerant state machine replication library
  • Written in Java, maintained and evolved during more than 10 years
  • Available under Apache license: http://bft-smart.github.io/library/
  • Implements a PBFT-like modular protocol

• Key features: modularity, reconfigurations, robustness, performance
BFT-SMaRt Performance (gigabit Ethernet, no disks)

Figure 4.6: Peak sustained throughput of BFT-SMaRt for CFT (2f + 1 replicas) and BFT (3f + 1 replicas) considering different workloads and group sizes.

Finally, it is also interesting to see that, with relatively big requests (1024 bytes), the difference between BFT and CFT tends to be very small, independently on the number of tolerated faults. Moreover, the performance drops between tolerating 1 to 3 faults is also much smaller with large payloads (both requests and replies).

Mixed workloads. Figure 4.7 reports the results of our experiment considering a mix of read and write requests. In the context of this experiment, the difference between reads and writes is that the former issues small requests (almost-zero size) but gets replies with payload, whereas the latter issues requests with payload but gets replies with almost zero size. This experiment was also conducted under a saturated system running 1600 clients. We performed the experiment both for the BFT and CFT setups of BFT-SMaRt, using requests and replies with payloads of 100 and 1024 bytes. Similarly to the previous experiments, the CFT protocol outperforms its BFT counterpart regardless of the ratio of read to write requests by around 5 to 15%.
Blockchain and Nakamoto Consensus

*The inspiration for most modern BFT protocols*
What is a Blockchain?

- Blockchain is a secure ledger of transactions implemented in a distributed (peer-to-peer) way.
- It requires solving consensus under Byzantine fault model.
  - Given a blockchain of \( i \) blocks, and several proposals for block \( i+1 \), how to decide (in a distributed way) which proposal to adopt?
Problem Statement

“A problem well stated is a problem half-solved.” - Charles Kettering

- Consider a system with \( n \) parties (honest or not) in which each one of them maintain its copy of the ledger

- A protocol \( P \) implements a robust public transaction ledger if it satisfies the following two properties:
  - **Persistence**: If an honest party reports a ledger that contains a transaction \( tx \) in a block more than \( k \) positions from the end of the ledger, then \( tx \) will be reported in the same position of the ledger by any other honest party
  - **Liveness**: If a “valid” transaction \( tx \) is sent to all honest parties, then there exist an honest party that will report \( tx \) at a block more than \( k \) blocks from the end of the ledger
Public (Open) Ledgers

• Originally implemented through Nakamoto Consensus or its variations

• Key ideas:
  • Anyone can participate in the network
  • A block can be added to the blockchain only if a cryptographic puzzle is solved
  • New blocks are disseminated in a peer-to-peer network
  • If multiple proposals for extending the chain are received, the longest proposal is used
Nakamoto Consensus
*(code on every peer)*

- **Local state:**
  - $C$: local copy of the blockchain

- **Algorithm:**
  - When a new chain $C'$ is received
    - $C = \text{maxvalid}(C, C')$
  - When a new batch of transaction $txs$ is received
    - $C = \text{proof-of-work}(C, txs)$
  - Broadcast($C$)
  - When a read request is received
    - Return the transactions on $C$

Compares two chains and chooses the longest one that is valid, i.e., each block is correctly signed, contains the hash of the previous and solved the proof-of-work puzzle.

Solves the following cryptopuzzle: find a valid block containing the transactions and the hash of the previous block such that the hash of this block is smaller than $D$ (a difficulty parameter).
Proof of Work

Cryptographic hash function (e.g., SHA256)

Message $m$ with any size

$h = \text{SHA256-HASH}(m)$

Hash $h$ w/ 32 bytes

• The Proof-of-Work is generated by changing the block until you find a hash that is smaller than a value depending on the block difficulty
  • E.g.: the first 76 bits (of 256) need to be zero

• Miners must try a lot, for example, in Bitcoin (17k nodes) it takes 10 min. on average
Why and how is it secure?

- The protocol works if the adversary controls less than half of the total computing power and the network disseminate data “fast enough”
Hybrid Consensus

- Permissionless blockchain protocols based on the use of both Nakamoto and “traditional” BFT consensus together
- We are particularly interested on the idea of using two blockchains:
  - One for defining a committee of members to participate in the ordering of transactions
  - Other, maintained by the committee, used for ordering transactions

- Observation:
  - This idea was introduced in Bitcoin-NG, in which two blockchains (with different PoW difficulties) were used
  - It is possible to define committee members without using a blockchain by requiring a PoW for participating for a time
Why Hybrid Consensus?

Throughput (txs/s) vs Number of nodes (n)

- Hybrid Consensus
- BFT Protocols
- Nakamoto Consensus

desired

achieved (due to the committee size requirements)
Hybrid Consensus: General Idea

Producers of last $\lambda$ blocks are the members of committee $R$

Each committee produces a “daily” blockchain with the transactions it validated

A block is validated by signatures of a committee quorum
Hybrid Blockchains

• This protocol achieves responsiveness, i.e., the commit latency of a transaction is proportional to the network latency
  • Nakamoto consensus is not responsive (latency depends on the time to solve the PoW) and BFT protocols are responsive (latency depends on network latency)

• Required assumptions (for responsiveness):
  • Synchronous system (as in Nakamoto consensus)
    • This work also shows that (1) permissionless consensus is impossible in partially synchronous systems, and (2) the peers need to continuously execute PoW for the snailchain
  • Less than 1/3 of Byzantine nodes
  • Corrupting a node takes some time \( \tau \)

• Practical observation: the size of the committee \( (\lambda) \) and the duration of the “day” are dependent of \( \tau \)
Chained Consensus
Blockchain-inspired consensus protocols

• Change of abstraction
(Non-chained) Consensus for Blockchains

- Consider a BFT SMR implementing a blockchain
  - V: validation of transactions
  - E: execution of transactions
- **ORDER-EXECUTE model** (traditional, e.g., SMaRtChain [DSN’20])
(Non-chained) Consensus for Blockchains

- Consider a BFT SMR implementing a blockchain
  - V: validation of transactions
  - E: execution of transactions
- EXECUTE-ORDER model (blockchain-oriented, e.g., SBFT [DSN’19])
Chained Consensus for Blockchains

- Chained consensus employ a sequence of weaker/simpler steps to notarize (or justify) blocks (batches) of transactions
  - Notarize/Justify: collect a certain number of signatures approving a block
- In the figure, when a block $i$ is voted by a quorum, block $i-1$ is committed
Streamlet

*Arguably the simplest BFT “consensus” protocol*
System Model

• $n$ nodes, with up to $t < n/3$ are subject to Byzantine failures
  • The static adversary “chooses” its $t$ corrupted nodes when the system starts

• Reliable communication channels

• Synchronized clocks
  • Basic time unit is called round (e.g., 10ms)

• Partial synchrony:
  • There is an unknown instant in the system execution, the *Global Stabilization Time* ($GST$), such that:
    • Before GST, the adversary can delay messages arbitrarily
    • After GST, messages sent by honest processes are received within $\Delta$ rounds
Streamlet Operation

• Streamlet runs in synchronized epochs, each $2\Delta$ long
• Each epoch is mapped to a random leader $L$ using a hash function $H$
  • There will be at most one block for each epoch
• Blockchain model:
  • Block $B = (h, e, tx)$
    • $h$ is the hash of the previous block
    • $e$ is the epoch number of the block
    • $tx$ is the set of transactions on the block
  • Genesis block: $(null, 0, null)$
  • A block is notarized if it is signed by more than $2/3$ of the nodes
  • A blockchain is notarized if all its blocks are notarized
Streamlet Protocol

• Epoch leader proposes its block
  • Example: H(1) = P₀, H(2) = P₂, H(3) = P₁, H(4) = P₀, ...
• Each node that accepts this block, vote for it with its signature for the block
• Epoch-e block is finalized (committed) if blocks for epochs e-1, e, e+1 are notarized
Streamlet Protocol Specification

The Streamlet blockchain protocol (<1/3 corrupt, partially synchronous)

For each epoch \( e = 1, 2, \ldots \):

- **Propose:** At the beginning of epoch \( e \), epoch \( e \)'s leader \( L \) does the following: let chain be (any one of) the longest notarized chain(s) that the leader has seen so far, let \( h := H^*(\text{chain}) \), and let txs be the set of unconfirmed pending transactions.

  The leader \( L \) sends to everyone the proposed block \( \{(h, e, \text{txs})\}_{pk_L} \) extending from the parent chain chain.

- **Vote:** During the epoch \( e \), every node \( i \) does the following. Upon receiving the first proposal \( \{(h, e, \text{txs})\}_{pk_L} \) from epoch \( e \)'s leader \( L \), vote for the proposed block iff it extends from one of the longest notarized chains that node \( i \) has seen at the time.

  To vote for the proposed block \( (h, e, \text{txs}) \), node \( i \) simply sends to everyone \( \{(h, e, \text{txs})\}_{pk_i} \).

- **Finalize:** On seeing three adjacent blocks in a notarized blockchain with consecutive epoch numbers, a node can finalize the second of the three blocks, as well as its entire prefix chain.
Streamlet Correctness

• Consistency

Lemma 2 (Main consistency lemma). *If some honest node sees a notarized chain with three adjacent blocks $B_0, B_1, B_2$ with consecutive epoch numbers $e, e + 1, e + 2$, then there cannot be a conflicting block $B \neq B_1$ that also gets notarized in honest view at the same length as $B_1$.*

• Liveness

Theorem 4 (Liveness). *After GST, suppose that there are 5 consecutive epochs $e, e + 1, \ldots, e + 4$ all with honest leaders, then, by the beginning of epoch $e + 5$, every honest node must have observed a new final block that was not final at the beginning of epoch $e$. Moreover, this new block was proposed by an honest leader.*

• Limitations of Streamlet:
  • Requires synchronized clocks
  • It is not responsive!

  A protocol is responsive, if its transaction confirmation time depends only on the network’s actual delay $\delta$, but not on any a-priori known upper-bound $\Delta$. 
CASPER (+ GHOST = GASPHER)

The foundation for Ethereum 2.0
Ethereum 2.0: the Gasper Protocol

- Gasper is the Proof-of-Stake (PoS) consensus protocol used in Ethereum 2.0
- The main reason to migrate from Nakamoto consensus (PoW) to Gasper (PoS) is the energy spending
- In a PoS protocol, miners (now called validators) commit cryptocurrency, not physical resources such as CPU to produce blocks
- In Gasper, instead of using CPU for creating PoW for new blocks, a participant pays to participate in block generation/validation (currently 32 ETH -> ~ €55,700)
- Gasper is based on two sub-protocols:
  - Casper the Friendly Finality Gadget (Casper FFG):
    used to mark certain blocks as finalized so that users can know for sure if a block belongs to the main chain or not.
  - The Latest Message Driven Greediest Heaviest-Observed Sub-Tree (LMD GHOST):
    a fork-choice rule
The Gasper Protocol in Two Slides

- The consensus participants are called validators $\mathcal{V} = \{v_1, v_2, ..., v_N\}$
- All messages are broadcasted using the P2P infrastructure of Ethereum
- The system is synchronous and time triggered:
  - Slot: some constant time slot (e.g., 12 seconds)
  - In each slot, at most one block can be added to the blockchain
  - Epoch: some constant number of $C$ slots (e.g., 32 slots = 6.4 minutes)
  - Blocks belonging to epoch $j$ have slot numbers $jC + k$, where $0 \leq k \leq C - 1$
The Gasper Protocol in Two Slides

• Operationally, Gasper works quite like Streamlet:
  • For each slot, a committee (with one leader) is randomly selected from $\mathcal{V}$
  • The slot leader can propose a block, validators vote for it
  • A block is justified if more than $2/3$ of the slot committee vote for it
  • A block $i$ is finalized if block $i + C$ is justified (see the figure)
  • Forks (due to misbehaviors or asynchrony) are solved using the LMD-GHOST rule
  • Misbehaviors are punished by slashing the validator committed stake
HotStuff

The state of the art in BFT protocol
System Model

• $n$ nodes, with up to $f < n/3$ subject to Byzantine failures

• Reliable communication channels

• Partial synchrony:
  • There is an unknown instant in the system execution, the *Global Stabilization Time (GST)*, such that:
    • Before GST, the adversary can delay messages arbitrarily
    • After GST, messages sent by honest processes to honest processes are received within $\Delta$ time units

• *No synchronized clocks*
HotStuff in a Nutshell

- HotStuff is a modern (responsive) protocol that promises simplicity and scalability way beyond PBFT

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Correct leader</th>
<th>Authenticator complexity</th>
<th>$f$ leader failures</th>
<th>Responsiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS [25]</td>
<td>$O(n^4)$</td>
<td>$O(n^4)$</td>
<td>$O(n^4)$</td>
<td></td>
</tr>
<tr>
<td>PBFT [20]</td>
<td>$O(n^2)$</td>
<td>$O(n^3)$</td>
<td>$O(f n^3)$</td>
<td>✓</td>
</tr>
<tr>
<td>SBFT [30]</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(f n^2)$</td>
<td>✓</td>
</tr>
<tr>
<td>Tendermint [15] / Casper [17]</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(f n^2)$</td>
<td></td>
</tr>
<tr>
<td>Tendermint$^<em>$ / Casper$^</em>$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(f n)$</td>
<td></td>
</tr>
<tr>
<td>HotStuff</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(f n)$</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

- HotStuff can be implemented in a chained way
- This simplifies the algorithm, making it quite compact (like Streamlet)
HotStuff protocol

- Core rules of the protocol:
  - Block \(b^*\) such that \(b^*.parent = b^*.\text{justify.node}\) (node here is a block) forms a **One-Chain** (1C)
  - If, additionally, \(b^*.parent = b''\) forms a One-Chain, we say \(b^*\) forms a **Two-Chain** (2C).
  - If, additionally, \(b''.parent = b'\) forms a One-Chain, we say \(b^*\) forms a **Three-Chain** (3C).
HotStuff in Action

One-Chain

Two-Chain

Three-Chain

b : v₃
b' : v₄
b'' : v₅
b* : v₆

Client

Request

Reply

P₀

P₁

P₂

P₃

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

Generic

G
Recent Results
Order Fairness

- This is problematic for some applications (e.g., decentralized finance) since tx ordering is completely under control of the leader.
- How to ensure a property like "if $tx_1$ is received before $tx_2$ in the majority of correct replicas, then $tx_2$ cannot be ordered before $tx_1$"
Order Fairness: Pompè [Zhang et al. OSDI’20]

- A new ordering phase is used to establish totally ordered timestamps for the commands.
- Then, it is safe to use any leader-based protocol because the leader must follow the timestamp order.
- There are several other works in this field, e.g., Themis (to appear at CCS’23).
BFT Scalability

- This is a huge topic, with lots of works
- Kauri [Neiheiser et al. SOSP’21] brings comm. complexity to $O(\log n)$
- Many other protocols (including some we discussed) employ a scalable P2P/gossip network for disseminating messages
- Ex: Narwhal [Danezis et al. EuroSys’22] decouples data dissemination from the consensus
Adaptive Fast Geo-Replication


Issues with Geo-Replication

• Different administrative domains
• Performance diversity
  • Across replicas
  • Across time
• Throughput can be improved with better networks
• Latency requires protocol *optimizations*
  • Speed of light is the network limit
  • Latency proportional to the roundtrip to a fast quorum
Classical BFT Replication

N=4, f=1

Egalitarian quorums,
Any 3 out of 4 replicas
**WHEAT**: WeigHt-Enabled Active replicaTion

- BFT-SMaRt extended for geo-replication
- Uses optimizations that lead to significant latency reduction:
  - Single leader in the best-connected site
  - Tentative executions (from PBFT)
  - Employs smaller quorums (weighted replication)

- Weighted replication: safe voting assignment scheme for SMR
  - Uses $\Delta$ extra replica(s) for quorum formation
  - Improves latency by enabling more choice upon quorum formation
  - Needs to preserve quorum intersection and tolerance to $f$ faulty replicas
Weighted BFT Replication

N=5, f=1, Δ=1 (extra)

Weighted quorums,
One set of 3 out of 5
and any set 4 out of 5
Weighted BFT Replication

- **Consistency**: All quorums that hold Q votes intersect by at least one correct replica
- **Availability**: There is always a quorum available in the system that holds Q votes

- **Safe minimality**: There exists at least one minimal quorum in the system
Weighted BFT Replication

Define the number of replicas $u$ that hold $V_{\text{max}} > 1$ votes, without violating $f$

CFT mode

\[
\begin{align*}
  n &= 2f + 1 + \Delta \\
  N_v &= \sum V_i = 2F_v + 1 \\
  Q_v &= F_v + 1 \\
  u &= f
\end{align*}
\]

BFT mode

\[
\begin{align*}
  n &= 3f + 1 + \Delta \\
  N_v &= \sum V_i = 3F_v + 1 \\
  Q_v &= 2F_v + 1 \\
  u &= 2f
\end{align*}
\]

Input: $f$ and $\Delta$

Output: $u$ and $V_{\text{max}}$

\[
V_{\text{max}} = \frac{\Delta + f}{f} = 1 + \frac{\Delta}{f}
\]
Size of fast quorums with different $f$ and $\Delta$

![Graph showing the size of fast quorums with different $f$ and $\Delta$.]

- $3f + 2$ ($\Delta = 1$)
- $4f + 1$ ($\Delta = f$)
- $5f + 1$ ($\Delta = 2f$)
- $6f + 1$ ($\Delta = 3f$)
AWARE: Adaptive Wide-Area REplication

The benefit of weighted replication depends on choosing an optimal weight configuration.

- The environment of the system (i.e., network characteristics) may change at runtime (e.g., due to a DDoS attack).

AWARE enables a geo-replicated consensus-based system to adapt to its environment!
AWARE Approach

• **Self-Monitoring**
  • AWARE uses reliable self-monitoring for adapting replicas’ voting weights and leader position at runtime

• **Self-Optimization**
  • AWARE continuously strives for consensus latency gains at runtime
  • Changes weights and leader location to minimize consensus latency
Evaluation of WHEAT and AWARE
Setup

• **AWARE** is implemented on top of WHEAT, which is based on BFT-SMaRt

• For evaluation, we use the **Amazon AWS cloud**, using EC2 instances of t2.micro type with 1 vCPU, 1 GB RAM and 8 GB SSD volume

• We select different regions for instances and use one client and one replica per instance

• Clients simultaneously send 1kB-requests across all sites
Clients’ Request Latency

Average latencies of all clients and 20 configurations

Observations

- The best configuration (<4,0>) performs about 39% faster than the median (<3,4>), 64% faster than the worst (<2,1>)
- Tuning voting weights can reduce latency (see configs. with the same leader)
- Leader relocation may be necessary for achieving optimal consensus latency
Runtime Behavior of AWARE

![Graph showing latency over time for different clients in various regions with labeled events such as System Start, Leader Change, and Network Perturbation]
Summary: WHEAT + AWARE

• **Ease of deployment**
  • AWARE provides the needed automation for finding an optimal configuration by tuning voting weights and/or relocating the leader

• **Adjust to varying conditions**
  • AWARE dynamically adjusts to changing conditions by shifting high voting power to replicas that are the fastest in a recent time frame

• **Compensate for faults**
  • AWARE detects (non-malicious) high-weight replicas failures and restores the availability of up to $f(V_{max} - V_{min})$ voting power by redistributing high weights

• **Ultimately, it is a way to deal with heterogeneity**
Chasing the Speed of Light

- A key observation of WHEAT/AWARE is that smaller consensus quorums can accelerate consensus.

- But the size of quorums depends on the configured resilience threshold $t$.

- **FlashConsensus**: threshold-adaptive BFT SMR for wide-area deployments
  - satisfying safety and liveness for $t < n/3$
  - achieves **fast commit latency** in an expected common-case, when there are no more than $t_{\text{fast}} = \lceil t/2 \rceil < n/6$ faulty replicas.
FlashConsensus: AWARE on Steroids

• Obtain smaller quorums through
  • tentative use of a lower resilience threshold ($t_{\text{fast}}$)
  • achieving threat level awareness through
    • the incorporation of abortable SMR [A]
    • and BFT forensic support [B]
  • Without sacrificing linearizability
    • but supporting speculation [C]


[C] Incremental consistency guarantees for replicated objects. OSDI 2016.
Experimental Evaluation on AWS

Figure 9: Achievable latency gains for the $n = 21$ AWS setup.

(b) Clients’ observed end-to-end latencies for protocol runs with BFT-SMaRt, AWARE and FLASHCONSENSUS. The client results are averaged over all regions per continent.
Experimental Evaluation: PBFT and HotStuff

Figure 13: Latencies of BFT-SMART and FlashConsensus for $n = 51$ replicas, observed from different client locations.

Figure 14: Latencies of HotStuff using FlashConsensus techniques for $n = 51$ replicas.
- For more details, please check out our paper!

- FlashConsensus combines **weighted replication**
  - with **abortable SMR** and BFT forensics
  - to safely underestimate the resilience threshold
  - faster quorums accelerate consensus decisions

- The potential for **latency speedup** is substantial
  - client-side speculation allows to further reduce latency
  - by relaxing consistency guarantees

---

**Chasing the Speed of Light:**
Low-Latency Planetary-Scale Adaptive Byzantine Consensus

Christian Berger\(^1\)  Livio Rodrigues\(^*\)  Hans P. Reiser\(^2\)  Vishnu Koga\(^*\)  Alysson Bessani\(^*\)
\(^1\)LISKG, Faculdade de Ciências / Universidade de Lisboa, Portugal
\(^2\)University of Pisa, Germany

Abstract
Blockchain technology has spurred renewed interest in planetary-scale Byzantine fault-tolerant (BFT) state machine replication (SMR). While recent works have mainly focused on improving the scalability and throughput of these protocols, few have addressed latency. We present FlashConsensus, a novel transformation for optimizing the latency of quorum-based BFT consensus protocols. FlashConsensus uses an adaptive resilience threshold that enables faster transaction ordering when the system contains few faulty replicas. Our construction exploits adaptive weighted replication to automatically assign high voting power to the fastest replicas, forming small quorums that significantly speed up consensus. Even when using such quorums with a smaller resilience threshold, FlashConsensus still satisfies the standard SMR safety and liveness guarantees with optimal resilience, thanks to the judicious integration of abortable SMR and BFT forensics techniques. Our experiments with tens of replicas spread in all environments show that FlashConsensus can order transactions with a latency of less than 0.4s, half the time of a PSBT-like protocol with optimal consensus latency in the same network, and reaching the latency of this protocol meaning on the theoretically best possible internet links (transmitting at 67% of the speed of light).

1 Introduction
State machine replication (SMR) is a general approach for achieving fault tolerance in distributed systems by coordinating client interactions with a set of (independent server) replicas [48]. As of recently, many scalable (BFT) SMR protocols have been proposed for usage in blockchain infrastructures, such as HotStuff [46], SBFT [30], Tendermint [13], Mir BFT [55], ResilientBC [27], DposPrPoLedger [49], and Karat [43]. These protocols employ either some dynamically elected leader [15, 30, 43, 46, 69], use multiple leaders [1, 75], or are leaderless [4, 21, 98].

Nevertheless, consensus in all these cases requires communication steps that involve a Byzantine majority of replicas under the assumption of a Byzantine adversary that controls up to a fixed resilience threshold of \(f = \lceil \frac{n}{3} \rceil\) replicas. Often, the quorum size for proceeding to the next protocol stage depends on this threshold; a Byzantine-as-abort consensus quorum with \(\lceil \frac{n}{3} \rceil\) replicas [90]. This size exploits roughly \(\frac{1}{3}\) of all replicas if an optimal resilience threshold is used.

As the go-replicated or planetary-scale systems, such as permissionless blockchains (e.g., [1, 21]) with tens of nodes distributed worldwide, a relevant optimization goal is lowering the end-to-end latency clients observe. Employing smaller quorums of closer replicas can significantly decrease SMR latency [34, 36]. The challenge is in using such smaller, faster quorums to ensure that they interact in sufficiently many replicas with all other quorums of the system. Such smaller, interacting quorums can be built using weighted replication, where faster replicas have more voting power. However, this approach requires more replicas than necessary for optimal resilience [94]. In fact, there is a trade-off between resilience and performance, as a smaller, faster quorums require more spare replicas [9].

Smaller quorums for better latency. To illustrate how a go-replicated system can progress faster by accessing a smaller quorum of replicas, we consider a weighted quorum [56] with \(w = 2\) replicas dispersed across all 25 AWS regions (see Figure 1a). When the system is configured for maximum resilience, it tolerates up to \(r = 6\) Byzantine replicas (the highest integer satisfying \(r < \frac{n}{3}\)) and has \(\Delta = 2\)-sparse replicas, while the smallest weighted consensus quorum \(Q^*\) contains 13 replicas (see [98] for details on these calculations). This number corresponds to only one replica less than using non-weighted replication. If we instead configure the system for tolerating \(r = 3\) failures, the smallest weighted quorum \(Q^*\) contains only 7 replicas, with \(\Delta = 1\). Furthermore, this quorum can be composed of closer replicas that can exchange votes with each other faster. This can occur swiftly proceeding through the stages of the consensus protocol (see Figure 1b), and ultimately leading to latency gains that clients around the globe can benefit from (see Figure 1c).
Avoiding Shared Vulnerabilities with Diverse Replication

State of the art

• PBFT paper by Castro & Liskov (OSDI’99):
  “Malicious attacks and software errors are increasingly common. ... can cause faulty nodes to exhibit Byzantine (i.e., arbitrary) behavior...” (Introduction)
  “We assume independent node failures.” (System Model)

• After that
  • Most works on BFT do not mention malicious attacks in their motivation (instead they talk about hardware errors and non-deterministic bugs)
  • Blockchain-motivated works target a security-relevant setting but rarely mention fault independence
Revisiting BFT for Security
Intrusion Tolerance – BFT replication

By using **equal replicas**, we are replicating the **same vulnerabilities**
Intrusion Tolerance – Diversity

The same attack/exploit does not work on diverse replicas
Intrusion Tolerance – Diversity

Given **enough time** the attacker will **compromise f+1 replicas**
Intrusion Tolerance – Recoveries

If the replicas are **periodically replaced**, the attacker's work will be harder
Our Goal

Attack effort proportional to $t$
LAZARUS

• A control plane that manages the diversity of a BFT system to improve its resiliency to malicious (intelligent) adversaries

• LAZARUS addresses two types of threats:
  • Newly found vulnerabilities affecting one or more replicas
    • Replicas are taken offline for security patching
  • Zero-day vulnerabilities affecting multiple replicas
    • Estimate the risk of replicas to have a common vulnerability in the future
LAZARUS Overview & Adversary Model

UNTRUSTED

R0 R1 R2 Rn
LTU LTU LTU LTU

BFT-Replicated service

Controller

Clients

Subject to Byzantine failures

TRUSTED

Small local trusted component that reboots the host

Logically-centralized orchestrator for reconfigurations

Not controlled by the adversary

OSINT

CVE Details

Logical Centralization Database
Replicas, Configurations, and Diversity Pool

- **Replica**: software stack for a replica
  - OS (distribution, not only the kernel)
  - Support software (JVM, DBMS, etc.)
  - Replication library
  - Service code

- **Configuration**: set of $n$ replicas

- **Replica pool**: set of available replicas from which you pick a configuration
How to predict the risk of common vulnerabilities using historical data?
Diversity-aware Reconfigurations

1. **Find past common vulnerabilities** on the replica pool
   - Fetch OSINT from NVD, ExploitDB, vendor sites, etc.
   - For each vulnerability, discover
     - which replicas in the pool are affected
     - exploit and patch information

2. **Measure vulnerability severity**

3. **Calculate the configuration risk**
   - Sum the severity of vulnerabilities affecting each pair of replicas from a config.

4. **Select the next configuration**
   - Will be one of the less risky that requires less changes
Effects of Diversity on BFT Performance
Diversity and Performance

### 18 Operating Systems

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>#Cores</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>UB14</td>
<td>Ubuntu 14.04</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>UB16</td>
<td>Ubuntu 16.04</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>UB17</td>
<td>Ubuntu 17.04</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>OS42</td>
<td>OpenSuse 42.1</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>FE24</td>
<td>Fedora 24</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>FE25</td>
<td>Fedora 25</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>FE26</td>
<td>Fedora 26</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>DE7</td>
<td>Debian 7</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>DE8</td>
<td>Debian 8</td>
<td>4</td>
<td>15GB</td>
</tr>
<tr>
<td>W10</td>
<td>Windows 10</td>
<td>4</td>
<td>1GB</td>
</tr>
<tr>
<td>WS12</td>
<td>Win. Server 2012</td>
<td>4</td>
<td>1GB</td>
</tr>
<tr>
<td>FB10</td>
<td>FreeBSD 10</td>
<td>4</td>
<td>1GB</td>
</tr>
<tr>
<td>FB11</td>
<td>FreeBSD 11</td>
<td>4</td>
<td>1GB</td>
</tr>
<tr>
<td>SO10</td>
<td>Solaris 10</td>
<td>1</td>
<td>1GB</td>
</tr>
<tr>
<td>SO11</td>
<td>Solaris 11</td>
<td>1</td>
<td>1GB</td>
</tr>
<tr>
<td>OB60</td>
<td>OpenBSD 6.0</td>
<td>1</td>
<td>1GB</td>
</tr>
<tr>
<td>OB61</td>
<td>OpenBSD 6.1</td>
<td>1</td>
<td>1GB</td>
</tr>
</tbody>
</table>

---

**Homogeneous setups**

**Diverse setups**

Throughput (ops/sec)

More in the paper
Confidential Byzantine Replication

Vassantlal, Alchieri, Ferreira, Bessani. **COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services.** IEEE SP’22 (Oakland).
Security Properties of Byzantine Replication

1. Send money to Bob
2. Process the payment
3. Notify

Availability
Integrity
Confidentiality

Alice
Bob
Confidentiality through Secret Sharing

1) Dealer divides the secret into shares

2) Each shareholder stores its share

3) Combiner reconstructs the secret by combining a subset of shares

Shamir’s scheme properties:
- Up to $t$ shares reveal nothing about the secret
- At least $t + 1$ shares are required to reconstruct the secret
Different Flavors of Secret Sharing

1) Shareholders and combiner cannot detect invalid shares

Dealer adds additional information in form of commitments that enables shareholders and combiner to verify validity of shares

2) Shareholders do not have how to recover shares

Shareholders send blinded shares to a recovering shareholder

3) The scheme is not safe against a mobile adversary

Shareholders periodically reshare their shares

Verifiable Secret Sharing Schemes

Proactive Secret Sharing Schemes

Dynamic Secret Sharing Schemes

\[
\begin{align*}
V_{old} & \quad \text{Resharing} \quad V_{new} \\
\frac{t}{n} & \quad \frac{t'}{n'}
\end{align*}
\]
COBRA – Confidential BFT Replication

COBRA DYNAMIC PROACTIVE SECRET SHARING (DPSS) SCHEME

- Modular protocol stack
- Recovery
- Reconfiguration
- Handles many secrets

Fundamental features required in a practical system

COBRA DPSS + BFT-SMaRt = Confidential Replication
COBRA – Confidential BFT Replication

COBRA adds a confidentiality layer into BFT SMR:

COBRA DPSS

- Used to generate random polynomials
- Protects the system against a mobile adversary and allows to reconfigure the system
- Used to recover replicas’ private state

BFT State Machine Replication

Recovery

Dynamic Resharing

Dist. Polynomial Generation

Byzantine Consensus

Update and read (share and combine private state)

Replica recovery (recover private state)

Group reconfiguration (reshare private state)

DPSS protocols described in the paper
COBRA Replica State Model

**Without confidentiality**

<table>
<thead>
<tr>
<th>ID</th>
<th>Pers. Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**With confidentiality**

<table>
<thead>
<tr>
<th>ID</th>
<th>Pers. Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S_1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Pers. Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S_2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Pers. Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S_3$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Pers. Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S_4$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**BFT SMR Secrecy**: An adversary learns no information about the confidential data, if the data is not accessible by faulty clients and the adversary controls no more than $t$ servers.
Beyond COBRA: Challenges

- **Programming**
  - Replicas have a common state and a secret state (shares or encrypted data)
  - For many applications, tx processing requires computing over encrypted data
  - How to integrate this model with a smart contract language like Solidity?

- **Programming**
  - How to avoid leaking access patterns?

- **Performance**

  Update Throughput
  (1kB txs, 10 replicas)

  - No confidentiality
    - VSSR [CCS ’19]
      - 481 tx/s
    - COBRA
      - 1761 tx/s
  - 8028 tx/s
  - 4071 tx/s

  No share verification on the replicas
Other topics I didn’t covered

• Randomized (asynchronous) protocols (e.g., Dumbo family)
  • Recent works match the throughput and latency(!) of leader-based protocols
• DAG-based protocols
  • BFT consensus using only consistent broadcast and some local rules
• Other leaderless protocols
  • RedBelly, ISS, Mir-BFT, etc.
• Hybrid protocols (or BFT with trusted components)
  • For matching crash fault tolerance resilience (2f+1 instead of 3f+1)
• BFT Protocol Forensics
  • For discovering who misbehaved in a protocol (used in FlashConsensus)
• Asymmetric Trust Protocols (e.g., Stellar payment network)
  • Not powerful enough to solve consensus with minimal knowledge
Questions?

- Alysson Bessani
  - anbessani@fc.ul.pt
  - www.di.fc.ul.pt/~bessani

This work is partially supported by FCT through the ThreatAdapt project (FCT-FNR/0002/2018), and the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).