Scaling Blockchains

Achieving Arbitrary Throughput and Sub-Second Latency

Introduction
A set of nodes
Byzantine Fault Tolerance

> \( \frac{2}{3} \)
1. make transaction
Blockchains

1. make transaction

2. submit transaction
Blockchains

1. make transaction
2. submit transaction
3. sequence and verify
Blockchains

1. make transaction
2. submit transaction
3. sequence and verify
4. store
The best example

1. Send 5 coins to Bob
2. Send 5 coins to Bob
3. Payment authorised?
4. Store
High latency
Slow finality
Low throughput
Hight Latency
Low Throughput
Outline

Part I: Increase throughput through sharding

Part II: Reduce latency with side infrastructures
Part I: Increase throughput
Key concepts of state sharding
Cross-shard consensus
Replay attacks against common systems
Secure Cross-shard consensus
Evaluation

Part II: Reduce latency
Part I: Increase throughput

Part II: Reduce latency

Blockchains for retail payments
FastPay as side infrastructure
Interfacing FastPay with a primary system
Implementation
Evaluation
Main Takeaways

Part I: Increase throughput
Key concepts of sharded distributed ledgers
Main challenges in building secure sharded systems in practice

Part II: Reduce latency
Side-infrastructures to bring blockchain-based payment systems to physical points of sales
How to integrate those infrastructures into a primary distributed ledger
What is not covered

- Privacy on blockchain
- Sybil resistance mechanisms
- Incentives of nodes operators
Increasing Throughput
with Sharded Blockchains

Part I
Scaling blockchains
Scaling blockchains
High throughput
BFT resilience
Fast finality
Linear scalability
The more machines you have, the bigger your throughput
State Sharding
State Sharding

shard 1

shard 2

shard 3
State Sharding
State Sharding
An example transaction

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
State Sharding
An example transaction

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
State Sharding

Only two acceptable final states

X₁
Shard 1

X₂
Shard 2

Y₁
Shard 1

Y₂
Shard 2

Y₃
Shard 3

Shard 3
Cross-Shard Consensus

How do shards communicate with each other?
When BFT fails, auditing mechanisms are in place to trace malicious and transparent processing of transactions within a decentralized infrastructure, and supports privacy features through modern smart contract and ecosystem platforms, such as Hyperledger Fabric [†]. Further, it provides ecosystem full nodes. Ethereum currently processes 4 transactions per second, out of theoretical maximum of 25. Furthermore, our experimental prototype shows that OmniLedger’s throughput scales linearly in the number of active validators, supporting Visa-level workloads and beyond, while confirming typical transaction commit latency via parallel intra-shard transaction processing, ledger pruning and correctness by using a bias-resistant public-randomness generator and rotating these assignments as the set of validators evolves.

This paper makes the following contributions:

• It provides a full implementation and evaluates the performance of OmniLedger, a new, secure, decentralized consensus protocol for distributed ledger systems, that supports high-integrity and high-privacy applications.

• It introduces a distinction between parts of the smart contract and the underlying distributed ledger, and only depend on their correctness, as well as the infrastructure that are trusted to maintain the integrity of their parties.

• It provides an alternative of apportioning consensus group membership based on validators’ computational power and efficiency, as its validator set. To support the more power-efficient alternative of proof-of-stake [†] and Hybrid Consensus [†], running a public validator set consisting of traditional ledgers, OmniLedger uses a sliding window of recent proof-of-work block miners to securely assign validators to shards, and of periodically producing a full implementation and evaluating the performance of a new, secure, decentralized consensus protocol for distributed ledger systems, that supports high-integrity and high-privacy applications.

• It demonstrates that it is possible to achieve both scalable and strongly bias-resistant consensus on traditional ledgers with a sliding window of recent proof-of-work block miners, and explores the bias-resistant public-randomness generator and its use in a decentralized consensus protocol.

• It addresses the second key security challenge of ensuring a negligible probability that any shard is compromised or proof-of-stake [†] and Hybrid Consensus [†], running a public validator set consisting of traditional ledgers, OmniLedger uses a sliding window of recent proof-of-work block miners to securely assign validators to shards, and of periodically producing a full implementation and evaluating the performance of a new, secure, decentralized consensus protocol for distributed ledger systems, that supports high-integrity and high-privacy applications.

• It chooses statistically representative groups of validators, each proportional to its computational power and efficiency, to securely assign validators to shards, and of periodically producing a full implementation and evaluating the performance of a new, secure, decentralized consensus protocol for distributed ledger systems, that supports high-integrity and high-privacy applications.

• It provides the first experimental evaluation of a large consensus group diversity in a real-world proof-of-work blockchain, and explores the bias-resistant public-randomness generator and its use in a decentralized consensus protocol.

The scalability of distributed ledgers (DLs), in both total scale-out and/or trade performance for security [†], introduces new security assumptions, alternative of apportioning consensus group membership based on validators’ computational power and efficiency, as its validator set. To support the more power-efficient alternative of proof-of-stake [†] and Hybrid Consensus [†], running a public validator set consisting of traditional ledgers, OmniLedger uses a sliding window of recent proof-of-work block miners to securely assign validators to shards, and of periodically producing a full implementation and evaluating the performance of a new, secure, decentralized consensus protocol for distributed ledger systems, that supports high-integrity and high-privacy applications.
Cross-Shard Consensus

Byzantine Agreement + 2-Phases Atomic Commit
Spoiler alert: Insecure under parallel composition
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]

client

shard 1

shard 2

shard 3
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
**S-BAC**

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]

- **client**
- **shard 1**
  - BFT
  - BFT
- **shard 2**
  - BFT
  - BFT
- **shard 3**
  - BFT

delete \( X_1, X_2 \); create \( Y_1, Y_2 \)
$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$
$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

**S-BAC**

Client

Shard 1

Shard 2

Shard 3

Phase 1

Phase 2
Atomix

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Cross-Shard Consensus
How does it achieve linear scalability?

The diagram shows a network of cross-shard consensus nodes (BFT) connected in a hierarchical structure. Each node (s1 to s6) is connected to the BFT consensus protocol, facilitating communication and consensus across different shards.
Insecure under parallel composition
Attacks

Double spend any object

- Does not need to collude with any node
- Acts as client or passive observer
- Re-orders network messages (not always needed)
Attack against S-BAC

Double-spend $X_1$

$$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$$
Attack against S-BAC

Double-spend $X_1$

$T'(\overline{x_1}, x_2) \rightarrow (y_1, y_2, y_3)$
Attack against S-BAC

Double-spend $X_1$

$T'(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

- $c$
- $s1$ → BFT
- $s2$ → BFT
- $s3$ →
Attack against S-BAC
Double-spend $X_1$

\[ T'(\overline{x_1}, x_2) \rightarrow (y_1, y_2, y_3) \]

lock $X_2$
Attack against S-BAC
Double-spend $X_1$

$T'(\tilde{x}_1, x_2) \rightarrow (y_1, y_2, y_3)$

$\text{c}$
$\text{pre-abort}(T')$
$\text{pre-accept}(T')$
$\text{lock } X_2$
Attack against S-BAC

Double-spend $X_1$

$T'(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c \quad \text{pre-abort}(T')$

$s1 \quad \text{BFT}$

$s2 \quad \text{pre-accept}(T')$

$s3 \quad \text{lock } X_2$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c \quad \text{pre-accept}(T)$

$s1 \quad \text{BFT}$

$s2 \quad \text{pre-abort}(T)$

$s3 \quad \text{lock } X_2$
Attack against S-BAC

Double-spend $X_1$

$T'(\overline{x_1}, x_2) \rightarrow (y_1, y_2, y_3)$

$c$  

$s1$  

$s2$  

$s3$  

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c$  

$s1$  

$s2$  

$s3$  

BFT  

pre-accept$(T)'$

pre-accept$(T)$

from shard 1

lock $X_2$

abort$(T)$

pre-abort$(T)$

pre-accept$(T)$

pre-abort$(T)'$

pre-accept$(T)'$
Attack against S-BAC

Double-spend $X_1$

$T'((x_1, x_2) \rightarrow (y_1, y_2, y_3))$

- $c$:
  - Pre-abort($T'$)
- $s_1$:
  - BFT
  - Pre-accept($T'$)
- $s_2$:
  - BFT
- $s_3$:
  - Lock $X_2$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

- $c$:
  - Pre-accept($T$)
  - Abort($T$)
- $s_1$:
  - BFT
- $s_2$:
  - Pre-abort($T$)
- $s_3$:
  - BFT

Abort($T'$)

Unlock $X_2$
Attack against S-BAC

Double-spend $X_1$

$T^*(x_1) \rightarrow (y_*)$

client

shard 1

BFT

10
Attack against S-BAC

Double-spend $X_1$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

client

shard 1

shard 2

shard 3

attacker
Attack against S-BAC
Double-spend $X_1$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

client

shard 1

shard 2

5

shard 3

attacker

pre-abort(T)

pre-accept(T)

pre-accept(T)
Attack against S-BAC

Double-spend $X_1$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

client

shard 1

shard 2

shard 3

attacker

pre-abort(T)

pre-accept(T)

pre-accept(T)
Attack against S-BAC
Double-spend $X_1$

Before attack

$X_1$  
10

$X_2$  
5

After attack

$Y^*$  
10

$Y_2$  
4

$Y_3$  
10
If it is not implemented, it does not work
Attacks against S-BAC

First phase
## Attacks against S-BAC
### First phase

Table 1: List of replay attacks against the first phase of S-BAC for all possible executions of the transaction `T`

<table>
<thead>
<tr>
<th>Phase 1 of S-BAC</th>
<th>Phase 2 of S-BAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shard 1 (potential victim)</td>
<td>Shard 2 (potential victim)</td>
</tr>
<tr>
<td>1</td>
<td>pre-accept(T) lock x₁</td>
</tr>
<tr>
<td>2</td>
<td>▶️pre-abort(T)</td>
</tr>
<tr>
<td>3</td>
<td>▶️pre-abort(T)</td>
</tr>
<tr>
<td>4</td>
<td>▶️pre-abort(T) ▶️pre-abort(T)</td>
</tr>
<tr>
<td>5</td>
<td>pre-abort(T) ▶️pre-accept(T) lock x₂</td>
</tr>
<tr>
<td>6</td>
<td>▶️pre-accept(T)</td>
</tr>
<tr>
<td>7</td>
<td>pre-accept(T) pre-abort(T) lock x₁</td>
</tr>
<tr>
<td>8</td>
<td>▶️pre-accept(T)</td>
</tr>
<tr>
<td>9</td>
<td>pre-abort(T) pre-abort(T)</td>
</tr>
</tbody>
</table>
Attacks against S-BAC
Second phase

client

shard 1

shard 2

shard 3
Attacks against S-BAC

Second phase

<table>
<thead>
<tr>
<th>Shard 1</th>
<th>Shard 2</th>
<th>Shard 3 (potential victim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 accept(T) create $y_1$; inactivate $x_1$</td>
<td>accept(T) create $y_2$; inactivate $x_2$</td>
<td>- create $y_3$</td>
</tr>
<tr>
<td>2 $\triangleright$accept(T)</td>
<td>$\triangleright$accept(T)</td>
<td>create $y_3$</td>
</tr>
<tr>
<td>3 $\triangleright$accept(T)</td>
<td>$\triangleright$accept(T)</td>
<td>create $y_3$</td>
</tr>
<tr>
<td>4 $\triangleright$accept(T)</td>
<td>$\triangleright$accept(T)</td>
<td>create $y_3$</td>
</tr>
<tr>
<td>5 abort(T) (unlock $x_1$)</td>
<td>abort(T) (unlock $x_2$)</td>
<td>-</td>
</tr>
<tr>
<td>6 $\triangleright$accept(T)</td>
<td>$\triangleright$accept(T)</td>
<td>create $y_3$</td>
</tr>
<tr>
<td>7 $\triangleright$accept(T)</td>
<td>$\triangleright$accept(T)</td>
<td>create $y_3$</td>
</tr>
<tr>
<td>8 $\triangleright$accept(T)</td>
<td>$\triangleright$accept(T)</td>
<td>create $y_3$</td>
</tr>
</tbody>
</table>
What causes these issues?

**Issue 1.** Input shards cannot associate protocol messages to a specific protocol execution.

**Issue 2.** Output shards (that are not also input shards) do not experience the first phase of the protocol
Easy Fix?

Global sequence numbers?  |  Wait for messages to arrive?

Linear Scalability  |  High Throughput
Byzcuit
Acknowledgments

Mustafa Al-Bassam  Alberto Sonnino  Bano Shehar  George Danezis

University College London
Fix issue 1

Add sequence numbers per object
Byzcuit
Fix issue 2

Dummy objects for output shards

$X_1, S_{X_1}$  
Shard 1

$X_2, S_{X_2}$  
Shard 2

$D_3, S_{D_3}$  
Shard 3
Byzcuit

\( \{S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3)\} \)
Byzcuit

\{S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3)\}

Check 1. Are all inputs active / transaction well formed?

Check 2. Is the sequence number $S_T \geq \max\{S_X_1, S_X_2\}$?
Byzcuit

\( \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \)

- **Client**
  - If checks fail: \( S_{X1} \leftarrow S_T + 1 \)

- **Shard 1**
  - **BFT**
  - If checks fail: \( S_{X2} \leftarrow S_T + 1 \)

- **Shard 2**
  - **BFT**
  - If checks fail: \( S_{D3} \leftarrow S_T + 1 \)

- **Shard 3**
  - **BFT**

- **TM**
Byzcuit

$$\{S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3)\}$$
Byzcuit

\( \{S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3)\} \)

**Diagram:**
- Client
  - Shard 1
    - BFT
  - Shard 2
    - BFT
  - Shard 3
    - BFT
  - TM
    - BFT
Byzcuit

\( \{S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3)\} \)
Byzcuit

\( \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \)

- **client**
- **shard 1**
- **shard 2**
- **shard 3**
- **TM**

If \((T, ST)\), inactivate \(X_1, X_2, D_3\) create \(Y_1, Y_2, Y_3\)
Why is Byzcuit secure?

**Issue 1.** Input shards cannot associate protocol messages to a specific protocol execution.

Sequence numbers: act as session ID

**Issue 2.** Output shards (that are not also input shards) do not experience the first phase of the protocol.

Dummy objects: all shards experience the first phase of the protocol.
Anyone can be a TM
Byzcuit
Implementation

• Fork of Java Chainspace
• Based on BFT-SMART
• Only a prototype to demonstrate its properties

https://github.com/sheharbano/byzcuit
Figure 7: The effect of the number of shards on transaction throughput. Each transaction has 2 input objects and 5 output objects, both chosen randomly from shards.

Figure 9: Client-perceived latency—the time from when a client submits a transaction, until it receives a decision from Byzcuit about whether the transaction has been committed—under varying system loads (expressed as transactions submitted to Byzcuit per second). We submit a total of 1200 transactions at 200–1000 transactions per second to Byzcuit with 6 shards. Each transaction has 2 inputs objects and 5 output objects, both chosen randomly from shards. When the system is experiencing a load of up to 1000 tps, clients hear back about their transactions in less than a second on average, even with our replay attack defenses.

8 Conclusion

We presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. These attacks affect both shard-driven and client-driven consensus protocols, and allow attackers to double-spend or lock objects with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest; most of the attacks work also under asynchrony. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental re-think of cross-shard commit protocols is required to protect against them.

We developed Byzcuit, a new cross-shard consensus protocol merging features from shard-led and client-led consensus protocols, and withstanding replay attacks. Byzcuit can be seen as unifying Atomix (from Omniledger) and S-BAC (from Chainspace), into an $O(n)$ protocol, that is efficient and secure. We implemented a prototype of Byzcuit and evaluated it on a real cloud-based testbed, showing that it is more performant than Chainspace, and on par with Omniledger performance. The resulting protocol is a drop-in replacement for either, and can be adopted to immunize systems based on those designs.
Figure 7: The effect of the number of shards on transaction throughput. Each transaction has 2 input objects and 5 output objects, both chosen randomly from shards.

Figure 8: Decrease of Byzcuit throughput with the number of dummy objects. Each transaction has 1 input object, and up to 5 dummy objects randomly selected from unique non-input shards. 6 shards are used.

Figure 9: Client-perceived latency vs. system load (number of transactions received per second by Byzcuit), for 6 shards with 2 inputs and 5 outputs per transaction (both chosen randomly from shards).

8 Conclusion

We presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. These attacks affect both shard-driven and client-driven consensus protocols, and allow attackers to double-spend or lock objects with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest; most of the attacks work also under asynchrony. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental re-think of cross-shard commit protocols is required to protect against them.

We developed Byzcuit, a new cross-shard consensus protocol merging features from shard-led and client-led consensus protocols, and withstanding replay attacks. Byzcuit can be seen as unifying Atomix (from Omniledger) and S-BAC (from Chainspace), into an \( O(n) \) protocol, that is efficient and secure. We implemented a prototype of Byzcuit and evaluated it on a real cloud-based testbed, showing that it is more performant than Chainspace, and on par with Omniledger performance. The resulting protocol is a drop-in replacement for either, and can be adopted to immunize systems based on those designs.
Conclusion
Part I - Increasing Throughput

Byzcuit

- S-BAC + Atomix
- High throughput, linear scalability, BFT resilience, Fast finality

- **Paper:** https://arxiv.org/abs/1901.11218
- **Code:** https://github.com/sheharbano/byzcuit
Reducing Latency
Using Side Infrastructures

Part II
What we have so far
from Part I

- High throughput
- Linear scalability
- BFT resilience
- Fast finality
What we have so far
and what is missing

Total Latency:
slowest shard during phase 1
+ 
slowest shard during phase 2
+ 
all communications
Make it practical for retail payment at physical points of sale

That is the ambition
What do we need?

Properties

What we want

• Low latency
• BFT reliance
• Fast finality
• High capacity

Current industry

?
Centralized systems
Slow Finality
Little capacity
In summary

What we want

- Low latency
- BFT reliance
- Fast finality
- High capacity

Current industry

- Low latency
- Centralized
- Slow finality
- Little capacity
FastPay

High-Performance Byzantine Fault Tolerant Settlement
FastPay
Acknowledgments

Mathieu Baudet
George Danezis
Alberto Sonnino

Facebook Novi
Overview

FastPay 1

Primary

FastPay 2

FastPay 3
Difference with blockchains

Blockchains

Byzantine Consensus

FastPay

Byzantine Consistent Broadcast
FastPay
How does it work?

sender

recipient
FastPay
How does it work?

1. transfer order
FastPay
How does it work?

1. transfer order
2. verify

sender

recipient
FastPay
How does it work?

1. transfer order
2. verify
3. signed transfer order
FastPay
How does it work?

1. transfer order
2. verify
3. signed transfer order
4. confirmation order
5. confirmation order
6. confirmation order
FastPay
How does it work?

1. transfer order
2. verify
3. signed transfer order
4. confirmation order
5. confirmation order
6. confirmation order
7. update
FastPay
Increasing capacity
FastPay
Interface it with a primary infrastructure

Smart Contract's state

- The committee information
- Total funds in the contract
- Last primary tx index
- "Redeem log"
FastPay
From primary infrastructure to FastPay

1. funding transaction
FastPay
From primary infrastructure to FastPay

1. funding transaction
2. synchronization order
FastPay
From primary infrastructure to FastPay

1. funding transaction
2. synchronization order
3. verify & update
FastPay
Interface it with a primary infrastructure

Smart Contract's state

• The committee information
• Total funds in the contract
• Last primary tx index
• "Redeem log"
FastPay
From the primary infrastructure to FastPay

1. transfer order

sender

smart contract
FastPay
From the primary infrastructure to FastPay

1. transfer order

sender

2. verify

smart contract
FastPay
From the primary infrastructure to FastPay

1. transfer order
2. verify
3. signed transfer order

sender
smart contract
FastPay
From the primary infrastructure to FastPay

1. transfer order
2. verify
3. signed transfer order
4. confirmation order

sender
smart contract
FastPay
From the primary infrastructure to FastPay

1. transfer order
2. verify
3. signed transfer order
4. confirmation order
5. update

sender

smart contract
FastPay
From the primary infrastructure to FastPay

1. transfer order
3. signed transfer order
4. confirmation order
6. redeem transaction
7. verify & update
FastPay
Implementation

• Written in Rust
• Networking: Tokio & UDP
• Cryptography: ed25519-dalek

https://github.com/calibra/fastpay
FastPay
Throughput Evaluation
FastPay
High concurrency

Figure 5: Variation of the throughput of transfer orders with the number of shards, for various levels of concurrency (in-flight parameter). The measurements are run under a total load of 1M transactions.

Figures 11 and 12 (see Appendix C) show the variation of the throughput of transfer and confirmation orders with the number of shards, for various total system loads—namely the total number of transactions in the test; they show that the throughput is not affected by the system load. The tests were performed with 4 authorities, and the client concurrency in-flight parameter set to 1,000. These figures illustrate that FastPay can process about 160,000 transactions per second even under a total load of 1.5M transactions, and that the total load does not significantly affect performance. These supplement figures 5 and 6 that illustrate the concurrent transaction rate (in-flight parameter) also does not influence performance significantly (except when it is too low by under-utilizing the system).

Readers may be surprised those measurements are key. The key measurement work by Han et al. [23] compares a number of permissioned systems under a high load, and shows that for all of Hyperledger Fabric (v0.6 with PBFT) [26], Hyperledger Fabric (v1.0 with BFT-Smart) [27], Ripple [15] and R3 Corda v3.2 [39] the successful requests per second drops to zero as the transaction rate increases to more than a few thousands transactions per second (notably for Corda only a few hundred). An important exception is Tendermint [10], that maintains a processed transaction rate of about 4,000 to 6,000 transactions per second at a high concurrency rate. Those findings were confirmed for Hyperledger Fabric that reportedly starts saturating at a rate of 10,000 transactions per second [34]. Our results demonstrate that FastPay continues to be very performant even under the influence of extremely high rates of concurrent transactions (in-flight parameter) and overall work load (total number of transactions processed), as expected. This is apparently not the norm.

Influence of the number of authorities. As discussed in Section 4, we expect that increasing the number of authorities only impacts the throughput of confirmation orders (that need to transfer and check transfer certificates signed by 2\(+1\) authorities), and not the throughput of transfer orders. Figure 7 confirms that the the throughput of confirmation orders decreases as the number of authorities increases. FastPay can still process about 80,000 transactions per second with 20 authorities (for 75 shards). The measurements are taken with an in-flight concurrency parameter set to 1,000, and under a load of 1M total transactions. We note that for higher number of authorities, using an aggregate signature scheme (e.g. BLS [8]) would be preferable since it would result in constant time verification and near-constant size certificates. However, due to the use of batch verification of signatures, the break even point may be after 100 authorities in terms of verification time.

7.3 Latency
We measure the variation of the client-perceived latency with the number of authorities. We deploy several FastPay multi-shard authorities on Amazon Web Services (all in Stockholm, eu-north-1 zone), each on a m5d.8xlarge instance. This class of instance guarantees 10Gbit network capacity, on a 3.1 GHz, Intel Xeon Platinum 8175 with 32 cores, and 128 GB memory. The operating system is Linux Ubuntu server 16.04. Each instance is configured to run 15 shards. The client is run on an Apple laptop (MacBook Pro) with a 2.9 GHz Intel Core i9 (6 physical and 12 logical cores), and 32 GB 2400 MHz DDR4.

The diagram shows the variation of the throughput of confirmation orders with the number of processes.
fn handle_cross_shard_commit(a, C) -> Result {
  let O = value(C);
  let recipient = match recipient(O) {
    Address::FastPay(recipient) => recipient,
    Address::Primary(_) => {
      bail!();
    }
  };
  ensure!(a.in_shard(recipient));
  let recipient_account = accounts(a).get(recipient).or_insert(AccountState::new);
  recipient_account.balance += amount(O);
  Ok()
}

fn handle_primary_synchronization_order(a, S) -> Result {
  /// Update recipient(S) assuming that S comes from a trusted source (e.g. Primary client).
  let recipient = recipient(S);
  ensure!(a.in_shard(recipient));
  if transaction_index(S) <= last_transaction(a) {
    /// Ignore old synchronization orders.
    return Ok();
  }
  ensure!(transaction_index(S) == last_transaction(a) + 1);
  last_transaction(a) += 1;
  let recipient_account = accounts(a).get(recipient).or_insert(AccountState::new);
  recipient_account.balance += amount(S);
  recipient_account.synchronized.push(S);
  Ok()
}

Figure 10: Authority algorithms for cross-shard updates and (Primary) synchronization orders.

Figure 11: Variation of the throughput of transfer orders with the number of shards, for various loads. The in-flight parameter is set to 1,000.

Figure 12: Variation of the throughput of confirmation orders with the number of shards, for various loads. The certificates are issued by 4 authorities, and the in-flight parameter is set to 1,000.
FastPay
Influence of the number of authorities

Figure 7: Variation of the throughput of confirmation orders with the number of authorities, for various number of shards. The in-flight parameter is set to 1,000 and the system load is of 1M transactions.

We observe that the client-authority WAN latency is low for both transfer and confirmation orders; the latency is under 200ms when the client is in the U.S. West Coast, and about 50ms when the client is in the U.K. Figure 8 illustrates the latency between a client creating and sending a transfer order to all authorities, and receiving sufficient signatures to form a transfer certificate (in our experiment we wait for all authorities to reply to measure the worse case where $f$ authorities are Byzantine). The latency is virtually constant as we increase the number of authorities, due to the client emitting orders asynchronously to all authorities and waiting for responses in parallel.

Figure 9 illustrates the latency to submit a confirmation order, and wait for all authorities to respond with a success message. It shows latency is virtually constant when increasing the number of authorities. This indicates that the latency is largely dominated by the network (and not by the verification of certificates). However, since even for 10 authorities a FastPay message fits within a network MTU, the variation is very small. Due to our choice of using UDP as a transport there is no connection initiation delay (as for TCP), but we may observe packet loss under very high congestion conditions. Authority commands are idempotent to allow clients to re-transmit to overcome loss without sacrificing safety.

Performance under failures. Research literature suggests permissioned blockchains based on (often leader-based) consensus suffer an enormous performance drop when some authorities fail. We measure the effect of authority failure in FastPay and show that latency is not affected when $f$ or fewer authorities are unavailable.

<table>
<thead>
<tr>
<th>$f$</th>
<th>Latency (ms ± std)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>43 ± 2</td>
</tr>
<tr>
<td>1</td>
<td>41 ± 3</td>
</tr>
<tr>
<td>2</td>
<td>44 ± 4</td>
</tr>
<tr>
<td>3</td>
<td>47 ± 2</td>
</tr>
</tbody>
</table>

Table 2: Crash-failure Latency.

We run our baseline experimental setup (10 authorities distributed over 10 different AWS instances), when a different number of authorities are not available for $f = 0, \ldots, 3$. We measure the latency experienced by a client on the same continent (Europe), sending a transfer order until it forms a valid transfer certificate. Table 2 summarizes the mean latency and standard deviation for different $f$. There is no statistically significant difference in latency, no matter how many tolerable failures FastPay experiences (up to $f \leq 3$ for 10 authorities). We also experimented with killing FastPay in /f.short influence of the number of authorities
FastPay
Latency

Figure 8: Variation of the latency of transfer orders with the number of authorities, for various locations of the client.

Figure 9: Variation of the latency of confirmation orders with the number of authorities, for various locations of the client.

We observe that the client-authority WAN latency is low for both transfer and confirmation orders; the latency is under 200ms when the client is in the U.S. West Coast, and about 50ms when the client is in the U.K.

We run experiments with the client in two different locations; (i) in the U.K. (geographically close to the authorities, same continent), and (ii) in the U.S. West Coast (geographically far from the authorities, different continent). Each measurement is the average of 300 runs, and the error bars represent one standard deviation; all experiments use our UDP implementation.

We measure the effect of authority failure in FastPay and show that latency is not affected when \( f \) or fewer authorities are unavailable.

Table 2: Crash-failure Latency.

<table>
<thead>
<tr>
<th>( f )</th>
<th>Latency (ms) ± std</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>43 ± 2</td>
</tr>
<tr>
<td>1</td>
<td>41 ± 3</td>
</tr>
<tr>
<td>2</td>
<td>44 ± 4</td>
</tr>
<tr>
<td>3</td>
<td>47 ± 2</td>
</tr>
</tbody>
</table>

Table 2 summarizes the mean latency and standard deviation for different \( f \). There is no statistically significant difference in latency, no matter how many tolerable failures FastPay experiences (up to \( f \leq 3 \) for 10 authorities). We also experimented with killing FastPay at various points and observed that the latency remains consistent.
Worst-case efficiency

**Blockchains**

Bad leader can slow down the protocol

**FastPay**

No leader, nothing changes
Simplicity favors robustness and performance
Making it simple is hard
FastPay
The cost of simplicity

- Less than 4,000 LOC
- Over 1,500 Git commits
- Took 2.5 months to 3 engineers
FastPay
Deployment costs

- AWS m5d.8xlarge instance
- ~ 5 USD / hour
Conclusion
Part II - Reducing Latency

FastPay

- Based on Byzantine Consistent Broadcast
- Simple design, low latency, high capacity, very robust

- **Code:** https://github.com/calibra/fastpay
Scaling Blockchains
Achieving Arbitrary Throughput and Sub-Second Latency

Conclusion
Outline

Part I: Increase throughput

Through sharding

Part II: Reduce latency

with side infrastructures
Cross-Shard Consensus
Tricky to implement right
Main Takeaways

Part I: Increase throughput

Key concepts of sharded distributed ledgers
Main challenges in building secure sharded systems in practice

Part II: Reduce latency

Side-infrastructures to bring blockchain-based payment systems to physical points of sales
How to integrate those infrastructures into a primary distributed ledger