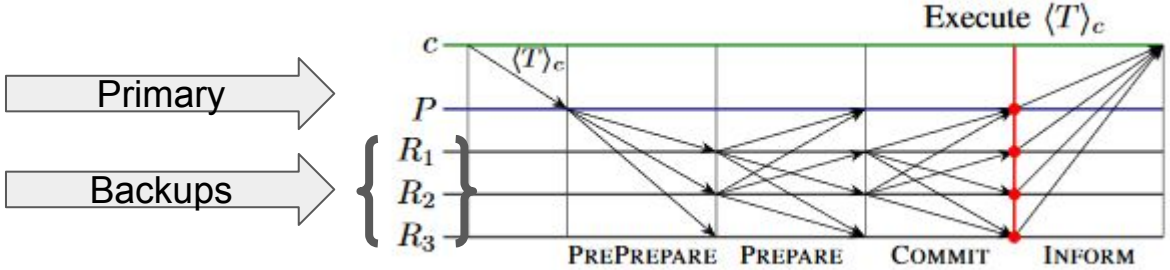
RCC: High-Throughput Secure Transaction Processing

Shawn Qiu, Di Zhao

Introduction

Resilient Concurrent Consensus is a paradigm used to increase throughput of consensus protocols such as *PBFT*, *HotStuff*, *ZYZZYVA, etc*

- Works with *Primary-Backup Consensus Protocols*



Notable authors: Mohammad Sadoghi

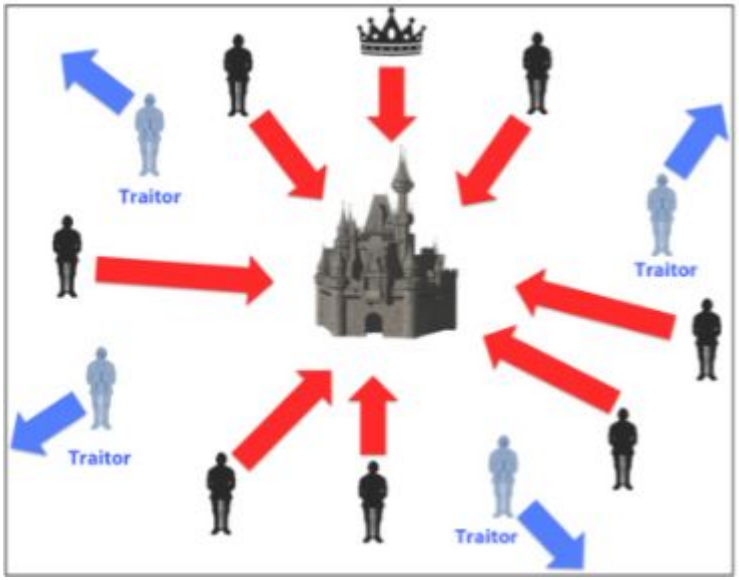
Paper is not yet published!

- Paper can be found on Slack and will be presented at ICDE ‘21

Benefits of Consensus-Based Systems

Benefits:

- More resilience during failures
- Support for data provenance
- Enables federated data processing



Challenges of Consensus-Systems

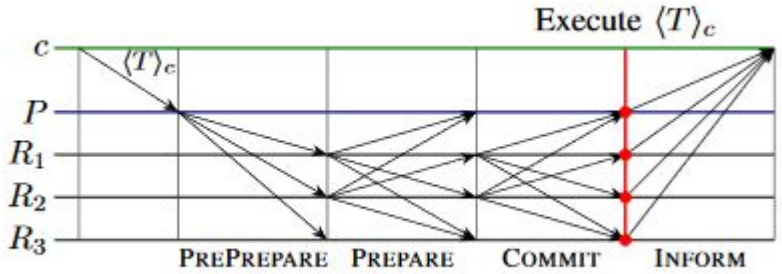
Performance

- We all want better performance!
- e.g. *ZYZZYVA* is an attempt to get better performance

RCC is a method of improving performance

Performance Challenges

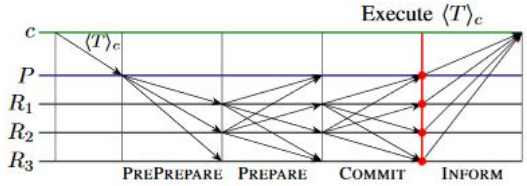
Specifically addresses the bottleneck of the primary's outbound bandwidth



The primary must send the *full transaction details* to **all** replicas while replicas only need to send messages to each other. In most implementations (via batching):

transaction size >> message size

Performance Bottleneck



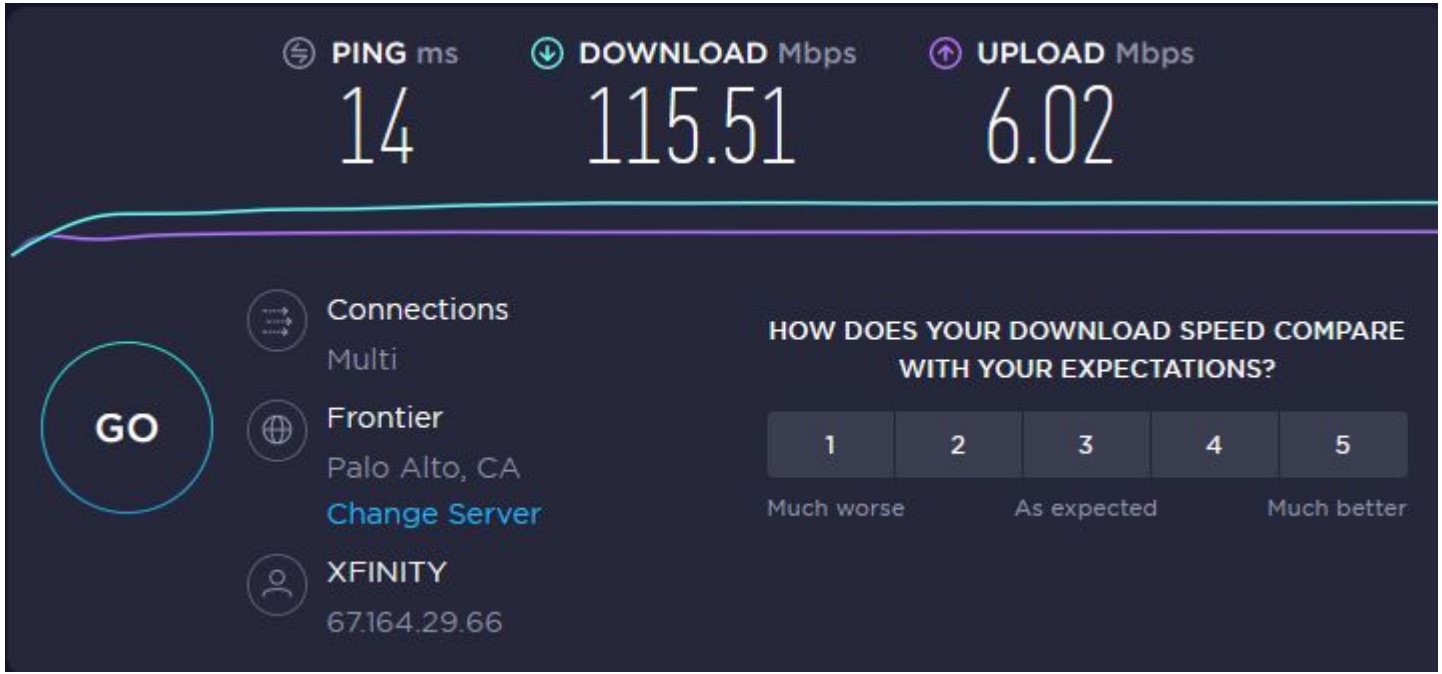
When transaction_size >> message_size, the amount of data sent/received by a machine:

Primary	**Replica**
(n-1)*transaction_size	transaction_size

Results:

1. Outgoing bandwidth of the primary is a performance bottleneck
2. Replicas are NOT fully utilizing all of their resources

Network Bandwidth



Additional Challenges

Primaries are heavily relied upon in traditional consensus protocols like PBFT. While they are resilient to failure in the primary, they are still vulnerable to the following

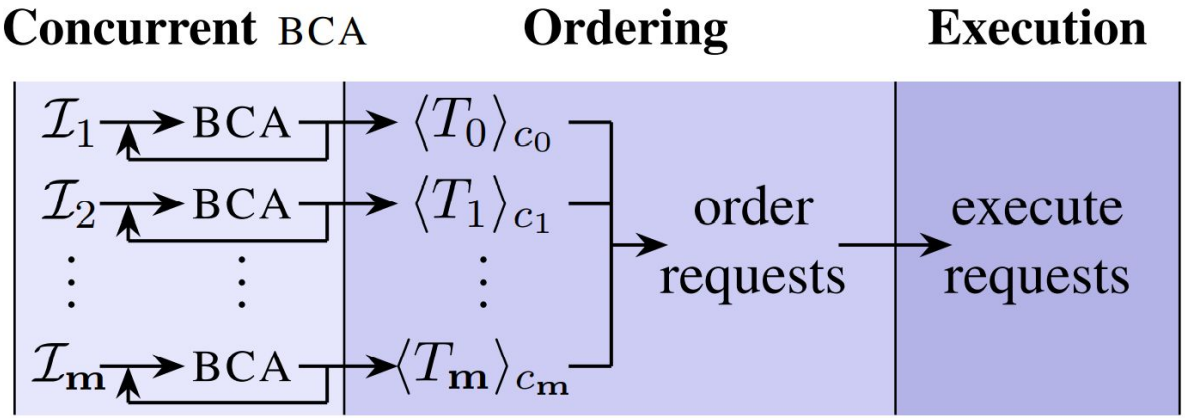
1. Ordering attacks
    a. Primary sets order for transactions. Can purposely reorder transactions for its own benefit
    b. e.g. Order checks and withdrawals from a bank account → force multiple overdrafts
2. Throttling attacks
    a. Slow throughput to just above the timeout rate
3. Targeted attacks
    a. DOS attacks on primary - slows throughput

Background/Terminology

- Consensus Protocols/Byzantine Consensus Algorithm (BCA)
  - PBFT, HotStuff, ZYZZYVA
- Byzantine commit
- Primary replacement
- Recovery
- deterministic

RCC Steps - High Level

1. Concurrent BCA - Byzantine Consensus Algorithm e.g. PBFT
2. Ordering - Order the transactions in a deterministic way
3. Execution - Execute request and notify client



Each replica is participating in m instances of BCA for which it is primary for one.

RCC - Basic Idea

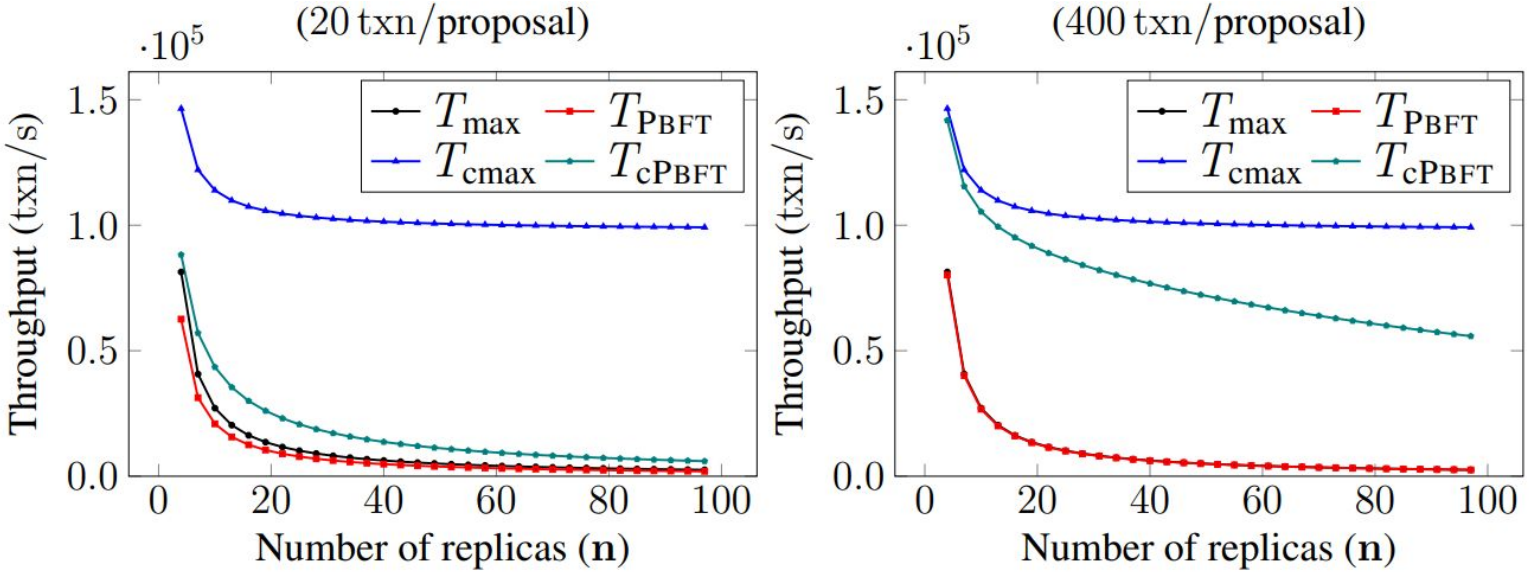
In overly broad terms, RCC's idea is simple:

1. Make every replica a "primary" node
2. Make every "primary" node handle different transactions simultaneously

Primary Challenges:

- Distribute client requests
- Handle detectable failures
- Handle undetectable failures

Performance Limits



RCC Solves Additional Challenges

1. Ordering attacks
   a. RCC proposes a method to deterministically order transactions
2. Throttling attacks
   a. A primary that falls behind in RCC will be detected as a failure
3. Targeted attacks
   a. Since there are many primaries, there isn't a single point to target. Primaries can also be load-balanced

RCC - Design Goals

1. RCC provides consensus among replicas on the client transactions that are to be executed and the order in which they are executed.
2. Clients can interact with RCC to force execution of their transactions and learn the outcome of execution.
3. RCC is a design paradigm that can be applied to any primary-backup consensus protocol, turning it into a concurrent consensus protocol.
4. In RCC, consensus-instances with non-faulty primaries are always able to propose transactions at maximum throughput (with respect to the resources available to any replica), this independent of faulty behavior by any other replica.
5. In RCC, dealing with faulty primaries does not interfere with the operations of other consensus-instances

Dealing with detectable failures:

1) All nf replicas need to detect failure

2) All nf replicas need to reach agreement on the state of faulty instance

3) All nf replicas need to determine which round the faulty primary allowed to resume its operations.

Recovery request role (used by replica R) **:**

1: **event** R detects failure of the primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m}$, in round ρ **do**
2: R halts $\mathcal{I}_i$.
3: Let P be the state of R in accordance to Assumption A3.
4: Broadcast FAILURE(i, ρ, P) to all replicas.
5: **event** R receives $\mathbf{f}+1$ messages $m_j = \text{FAILURE}(i, \rho_j, P_j)$ such that:
1) these messages are sent by a set S of $|S| = \mathbf{f}+1$ distinct replicas;
2) all $\mathbf{f}+1$ messages are well-formed; and
3) ρ_j, $1 \leq j \leq \mathbf{f}+1$, comes after the round in which $\mathcal{I}_i$ started last
do
6: R detects failure of $\mathcal{P}_i$ (if not yet done so).

Recovery leader role (used by leader $\mathcal{L}_i$ of P) **:**

7: **event** $\mathcal{L}_i$ receives $\mathbf{nf}$ messages $m_j = \text{FAILURE}(i, \rho_j, P_j)$ such that
1) these messages are sent by a set S of $|S| = \mathbf{f}+1$ distinct replicas;
2) all $\mathbf{nf}$ messages are well-formed; and
3) ρ_j, $1 \leq j \leq \mathbf{f}+1$, comes after the round in which $\mathcal{I}_i$ started last
do
8: Propose $\texttt{stop}(i; \{m_1, \ldots, m_{\mathbf{nf}}\})$ via P.

State recovery role (used by replica R) **:**

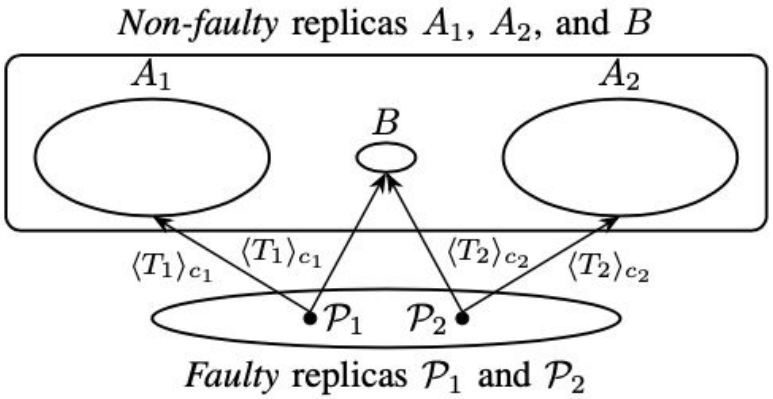
9: **event** R accepts $\texttt{stop}(i; E)$ from $\mathcal{L}_i$ via P **do**
10: Recover the state of $\mathcal{I}_i$ using E in accordance to Assumption A3.
11: Determine the last round ρ for which $\mathcal{I}_i$ accepted a proposal.
12: Set $\rho + 2^f$, with f the number of accepted $\texttt{stop}(i; E')$ operations, as the next valid round number for instance $\mathcal{I}_i$.

Dealing with undetectable failures:

Run a standard checkpoint algorithm for each BCA instance:

1) checkpoint algorithms can be run concurrently with the operations of BCA instances
2) only perform checkpoints after every x-th round for some system-defined constant x
3) Perform checkpoints on a dynamic *per-need basis*

in-the-dark attacks



Handling Client Requests

Distribute Requests :

- every instance proposes distinct client transaction

Forcing faulty primary to respond :

- Primaries do not receive client requests
- Faulty primaries that refuse to propose requests of some client
- Primaries are unwilling or incapable of proposing requests of some client

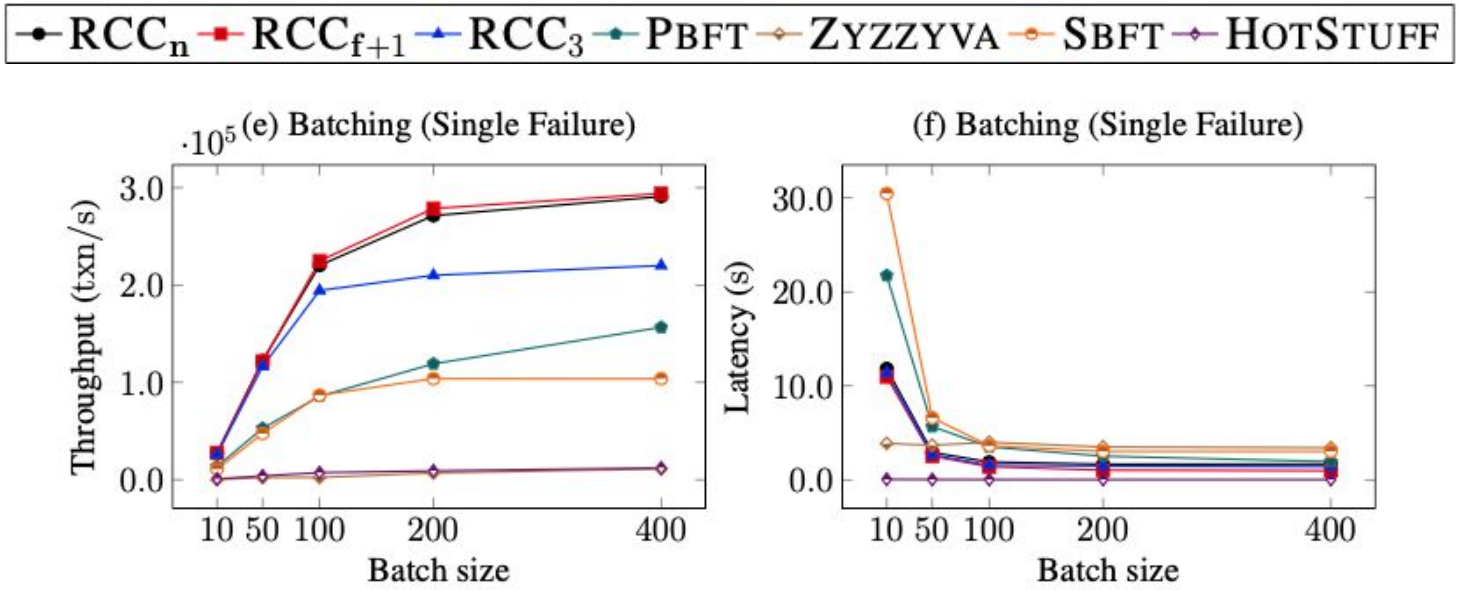
Benchmarks

Questions to answer :

- Performance : RCC provides more throughput than any primary-backup consensus protocol ?
- Scalability : RCC provides better scalability ?
- Concurrency : Does RCC provide sufficient load balancing of primary tasks ?
- Failure capacity : How does RCC fare under failures ?
- Batch client transaction : Impact of batching client transactions ?

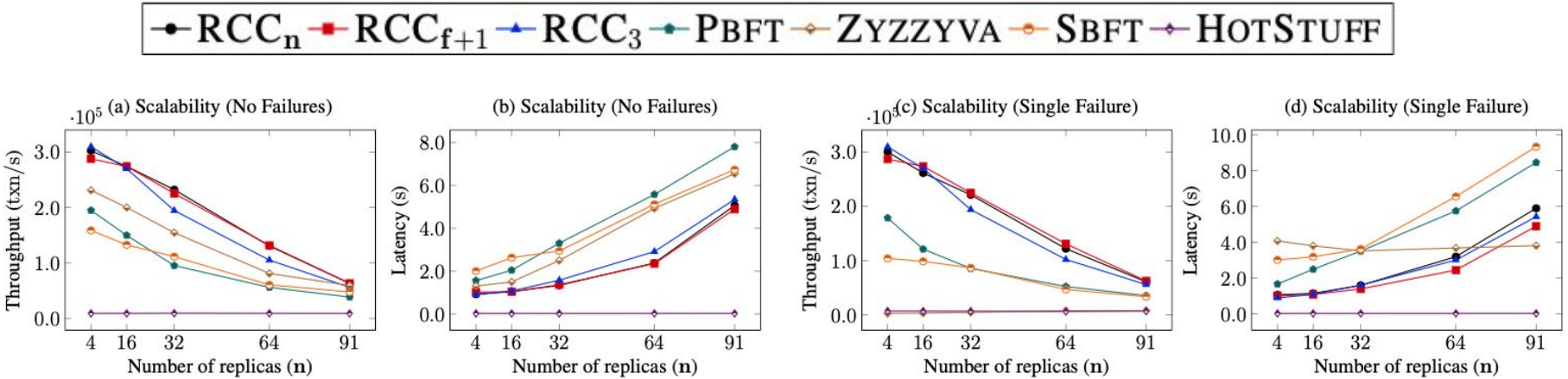
Benchmark

- Performance increase when batch size increase
- Chosen to use 100 txn/batch in all other experiments



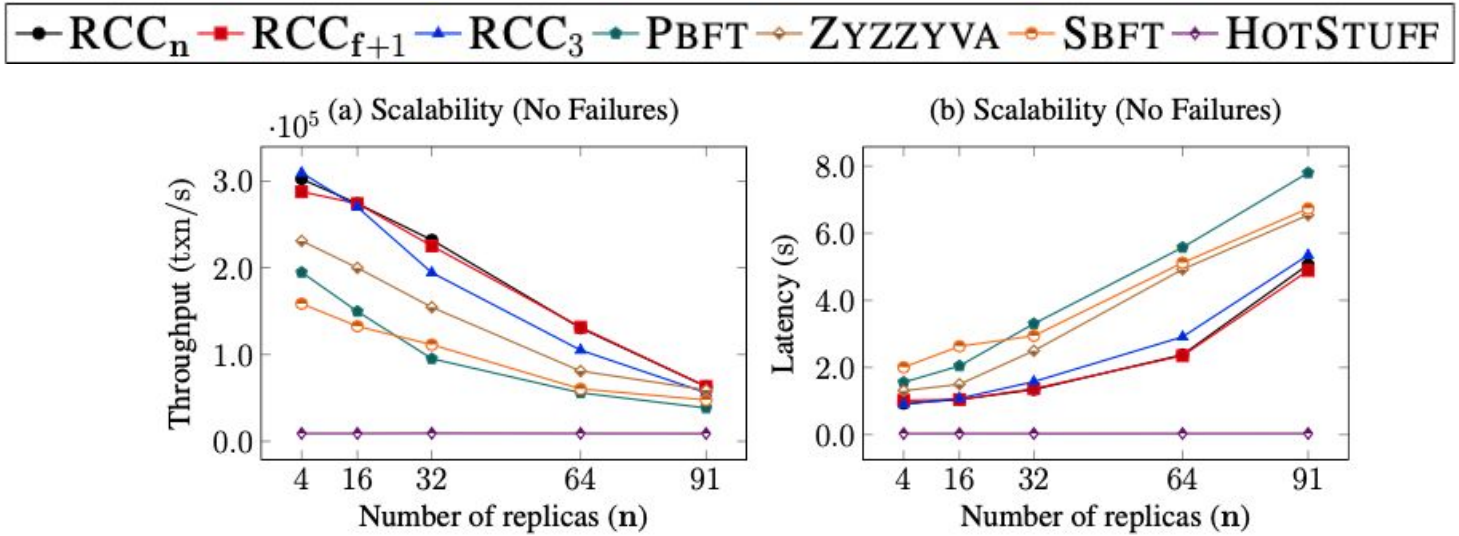
Benchmark

- Three versions of RCC outperform all other protocols
- Adding concurrency by adding more instances improves performance
- On small deployments with n = 4, . . . , 16 replicas, the strength of RCC is most evident



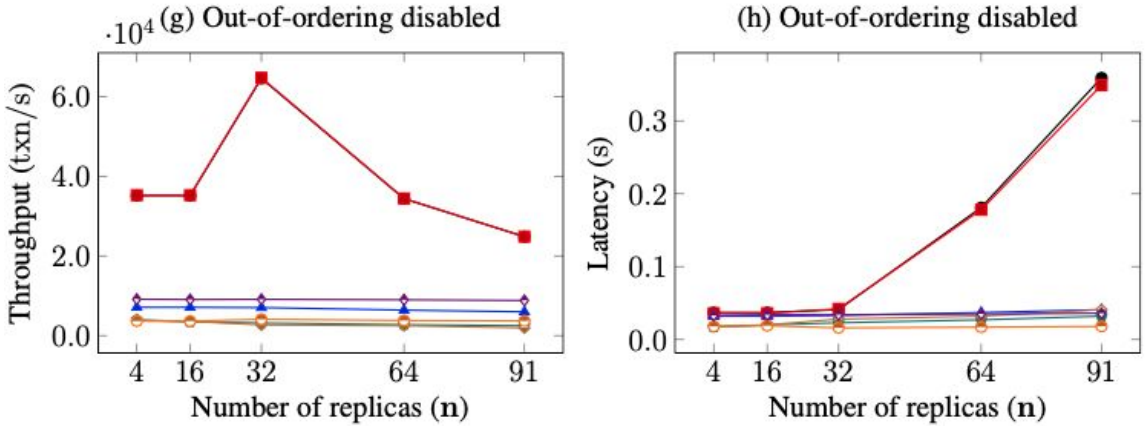
Benchmark

- RCC easily outperforms ZYZZYVA, even in the best-case scenario of no failures
- ZYZZYVA is—indeed—the fastest primary- backup consensus protocol when no failures happens



Benchmark

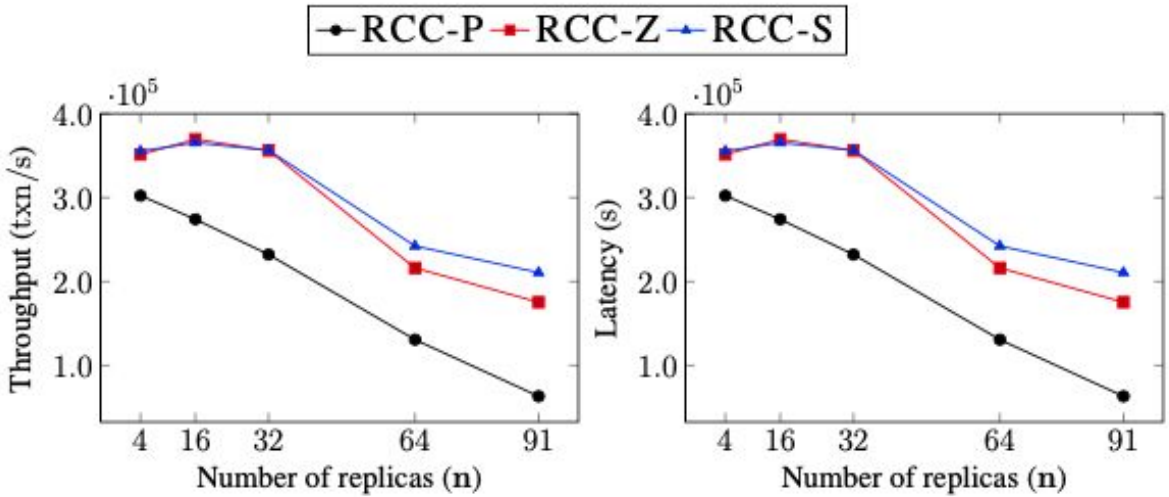
- HOTSTUFF event-based single- phase design outperforms all other primary-backup consensus protocols
- a non-out- of-order-RCC is still able to greatly outperform HOTSTUFF
- RCCf+1 and RCCn benefit from increasing the number of replicas



Benchmark

- RCC-S consistently attains equal or higher throughput than RCC-Z
- ZYZZYVA scales better than SBFT

- RCC-P (RCC+PBFT)
- RCC-Z (RCC+ZYZZYVA)
- RCC- S (RCC+SBFT)



Results

- Increasing the batch size increases performance of all consensus protocols
- three versions of RCC outperform all other protocols
- the ability of RCC, and of concurrent consensus in general, to reach throughputs no primary-backup consensus protocol can reach
- a non-out- of-order-RCC is still able to greatly outperform HOTSTUFF

Weaknesses

Client does not know the transactions that will come before or after it

E.g.

Client 1 tries to move $5 from account A to account B

Client 2 tries to move $5 from account A to account C simultaneously (same cycle)

If account A only has $5, client 2’s request will fail

Questions?