RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing

Presenter: Justin Xu, Shuxian Zhang, Wenchang Liu, Zhixuan Li

Content

- Motivation of RCC
- High level overview and normal case example of RCC
- Dealing with failures
- Client Interactions
- Evaluation of RCC

Limitations of Traditional Consensus

Traditional primary-backup consensus protocols underutilize network resources and thus prevents maximization of transaction throughput.

- Transaction throughput is determined by outgoing bandwidth **B** of primary.

$$T_{max} = \frac{B}{((n-1)\mathbf{st})} \qquad T_{PBFT} = \frac{B}{((n-1)(\mathbf{st} + 3\mathbf{sm}))}$$

st is the size of each transaction

sm is the size of each message

Limitations of Traditional Consensus (cont.)

$$T_{PBFT} \approx T_{max} \text{ when } \mathbf{st} >> \mathbf{sm}$$

- Replicas are underutilized: primaries must send (n-1)**st** while replicas only have to send and receive **st** bytes roughly, given that **st** >> **sm**.

- Underutilization of non-primary replicas in comparison to primaries!

Promise of Concurrent Consensus

Democracy - Give all the replicas the power to be the primary.

Parallelism - Run multiple parallel instances of a BFT protocol.

Decentralization - Always there will be a set of ordered client requests.

Promise of Concurrent Consensus

- HotStuff balances load by consistently switching primaries, but does not address core issue of underutilization of resources.

- Concurrent consensus involves proposing at least nf transactions at a time.

$$T_{cmax} = \mathbf{nf}\frac{B}{((\mathbf{n}-1)\mathbf{st} + (\mathbf{nf}-1)\mathbf{st})}$$

Towards Resilient Concurrent Consensus

- Concurrent consensus can achieve higher levels of throughput by efficiently utilizing all available replicas.

- RCC is a paradigm for transforming any primary-backup consensus protocol into a concurrent consensus protocol with increased throughput.

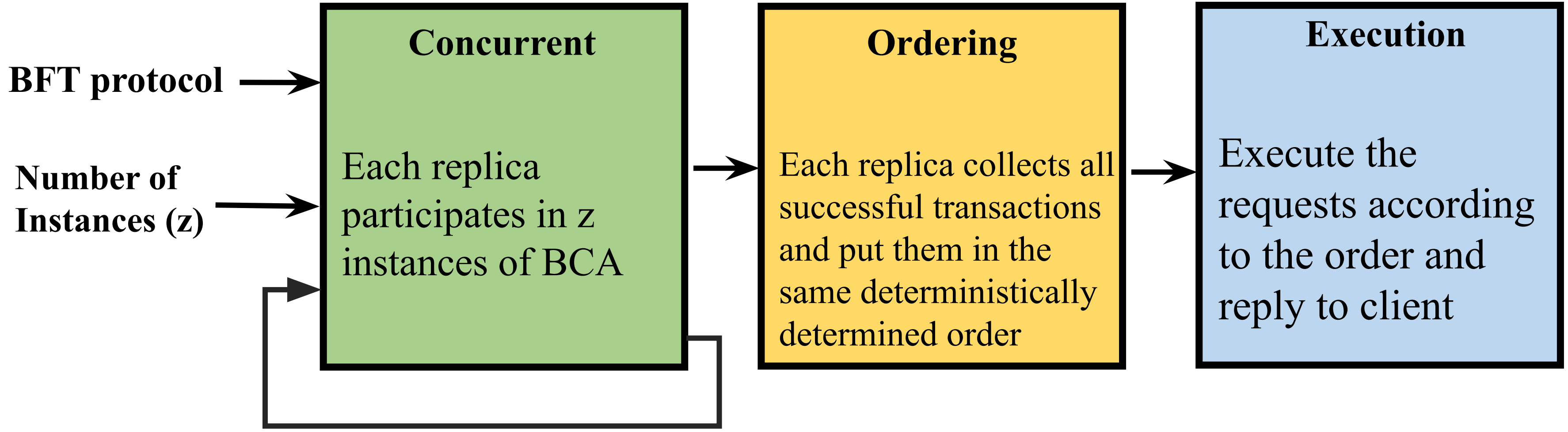
Basic Idea and Design Goals of RCC

- **Idea: Increase throughput by concurrency**
  - Making every replica a primary node
  - Primary nodes propose transactions simultaneously
- **Design Goals:**
  - Provide consensus among replicas on client transactions
  - RCC is a paradigm that can be applied to any other protocols
  - Non-faulty primaries can reach maximum throughput without being affected by faulty behaviors from other replicas
  - Dealing with faulty primaries does not interfere with the operations of other consensus-instances
  - Clients can interact with RCC to force execution of their transactions and learn the outcome of execution

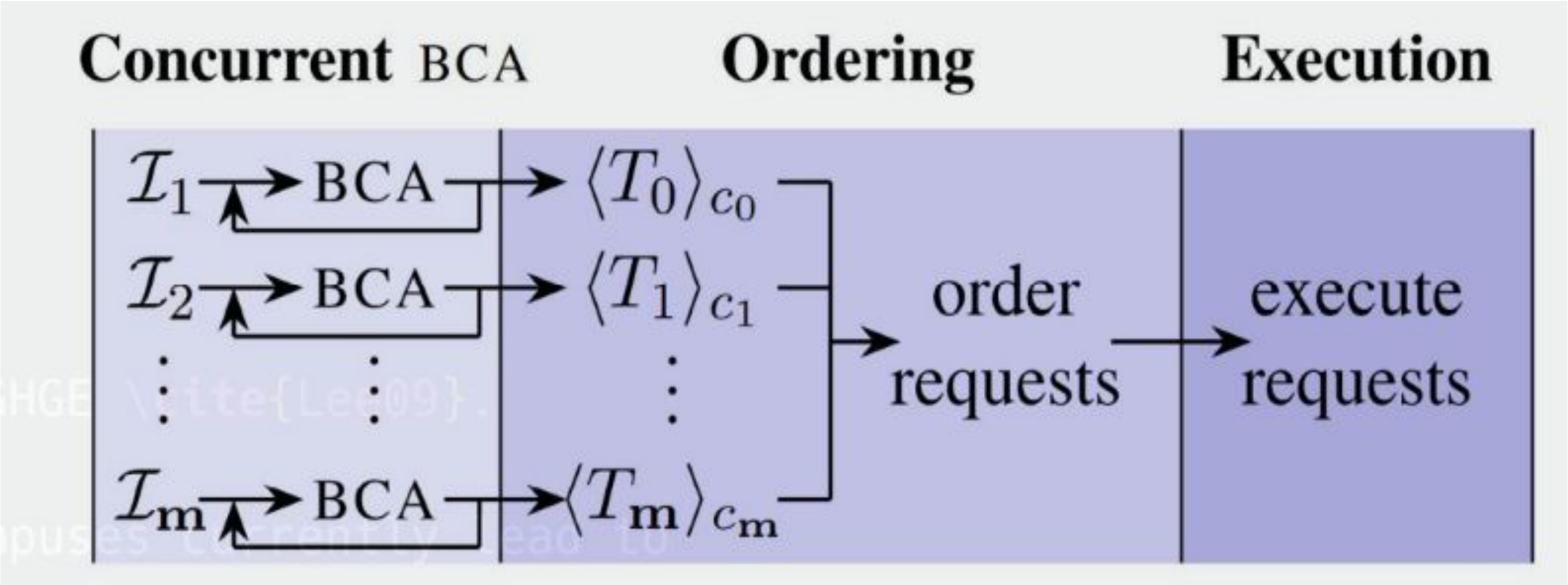
Background / Terminology

- **BCA:**
  - Byzantine commit algorithm(e.g. PBFT, Zyzzyva, HotStuff etc.)
- **Deterministic:**
  - identical inputs -> identical outputs

RCC Paradigm (Each round)



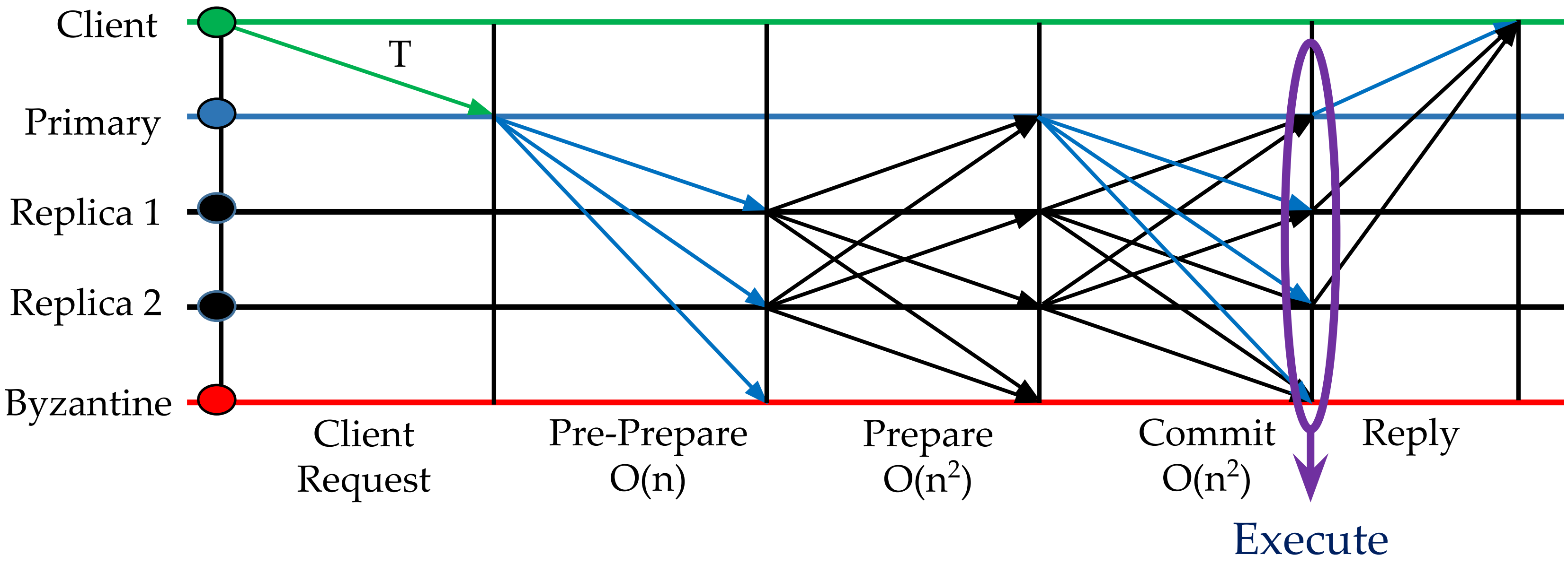
RCC Paradigm (High Level)



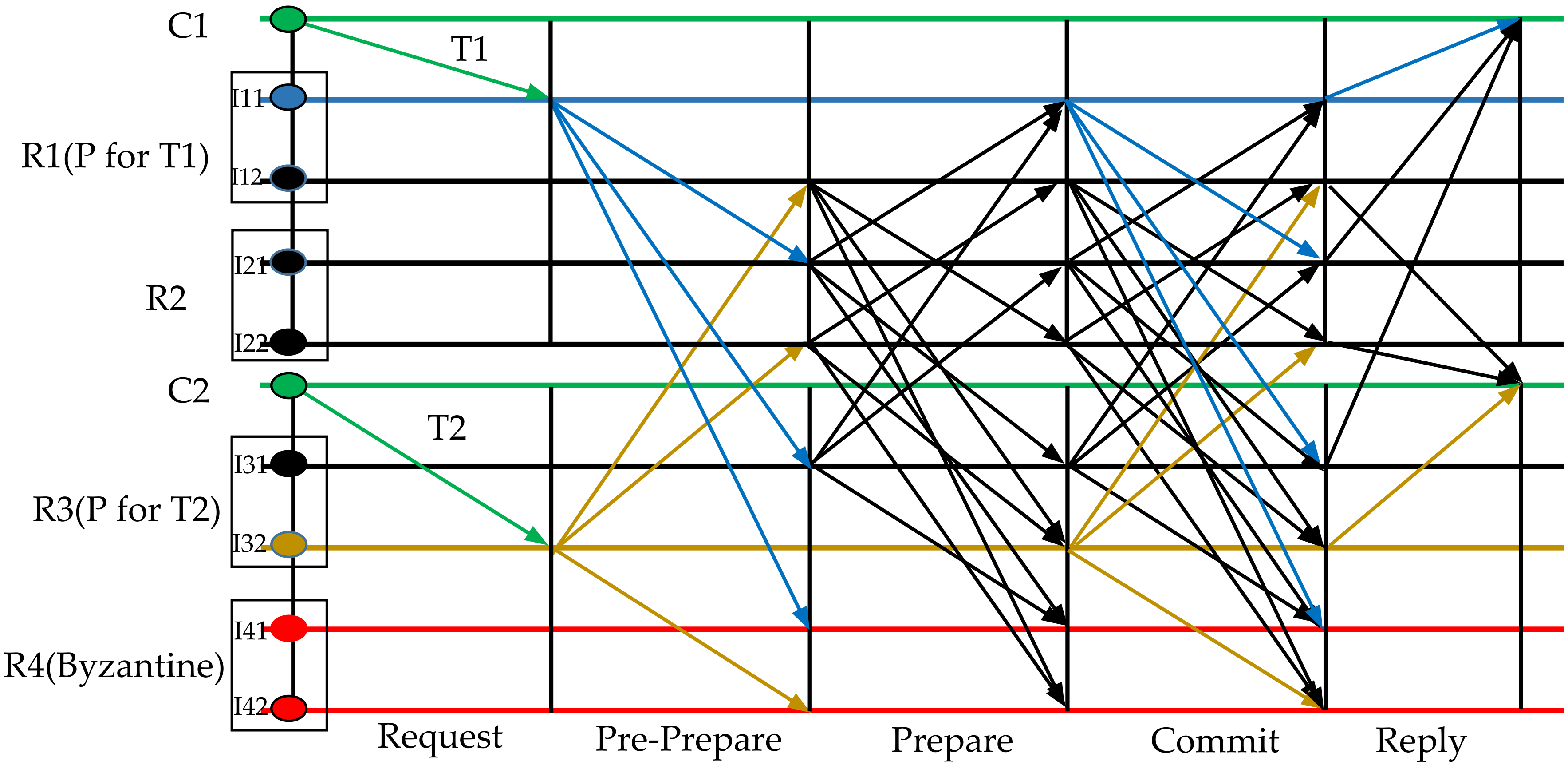
If we have m transactions and m replicas:

- Each replica in each round will have **m** instances participate in **m** BCA.
- Each replica can have one instance to be the primary of a BCA.

Practical Byzantine Fault Tolerance (PBFT)

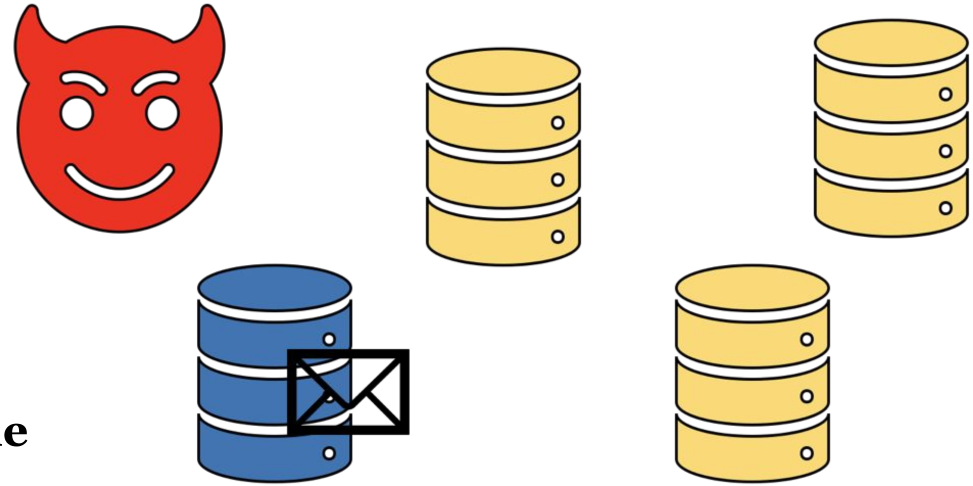


Normal Case Example: RCC using PBFT with 2 parallel instances



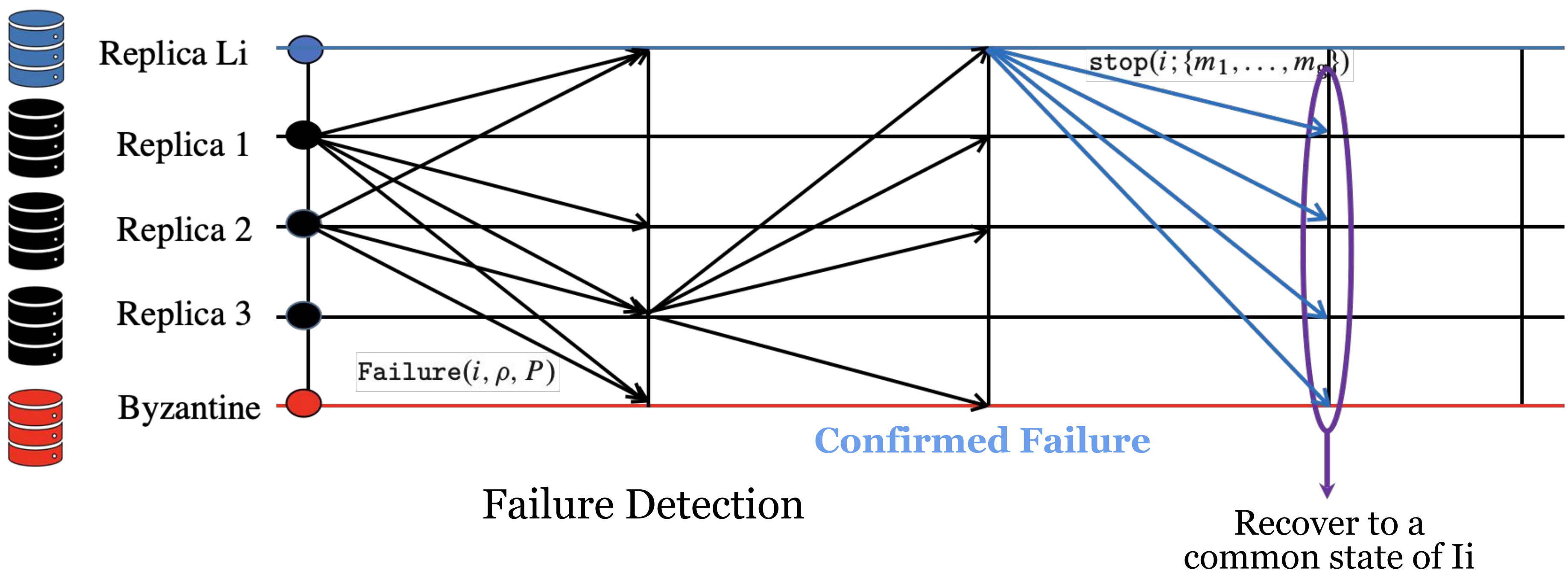
Note: Instance Iij for Replica i participates in Transaction Tj (i.e. I11,I21,I31,I41 -> T1, I12,I22,I32,I42 -> T2)

Dealing with Failures



- **Detectable**
- **Undetectable**

Dealing with Detectable Failures

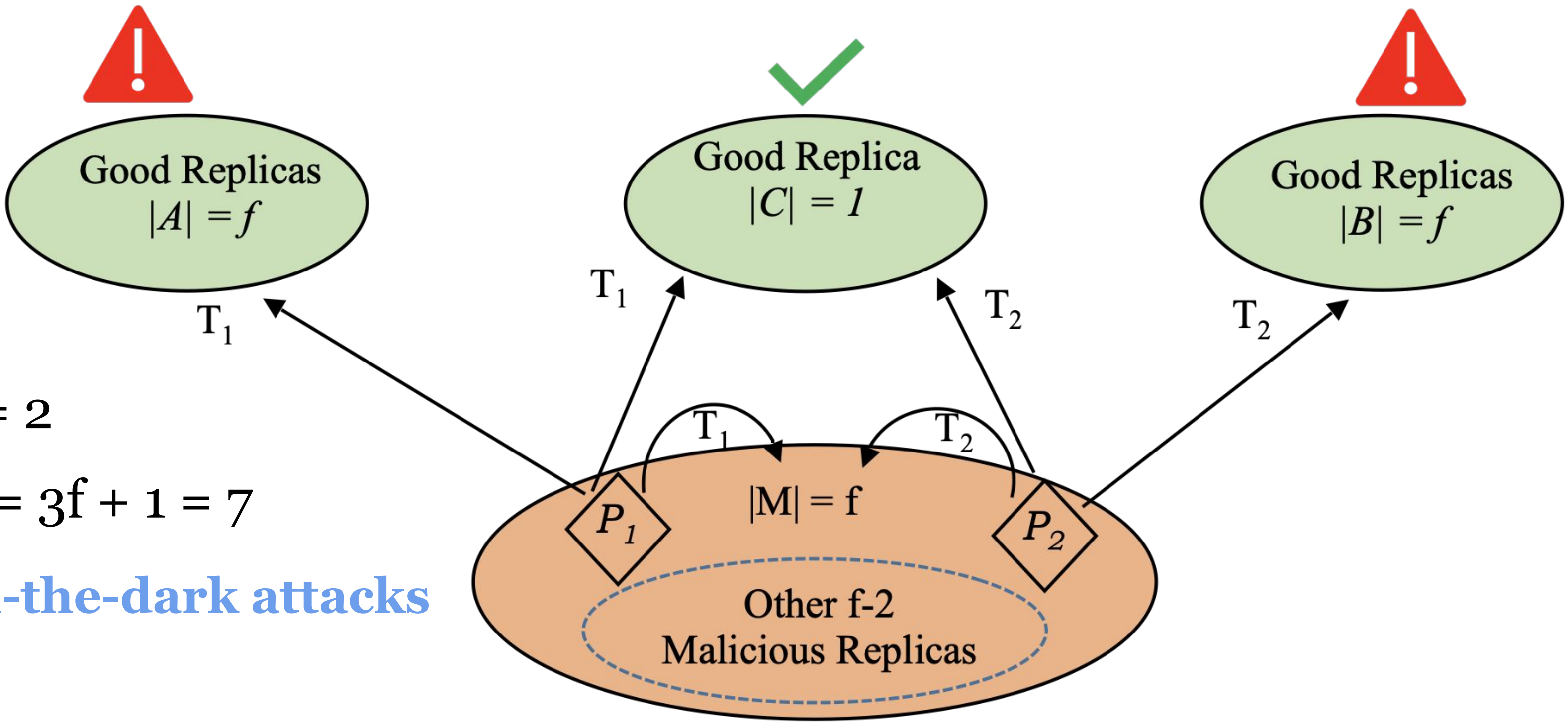


Dealing with Detectable Failures

The **recovery process** in 3 steps:

- All nf replicas need to detect failure
- All nf replicas need to reach agreement on the state of faulty instance
- All nf replicas need to determine which round the faulty primary allowed to resume its operations.

Dealing with Undetectable Failures



$f = 2$

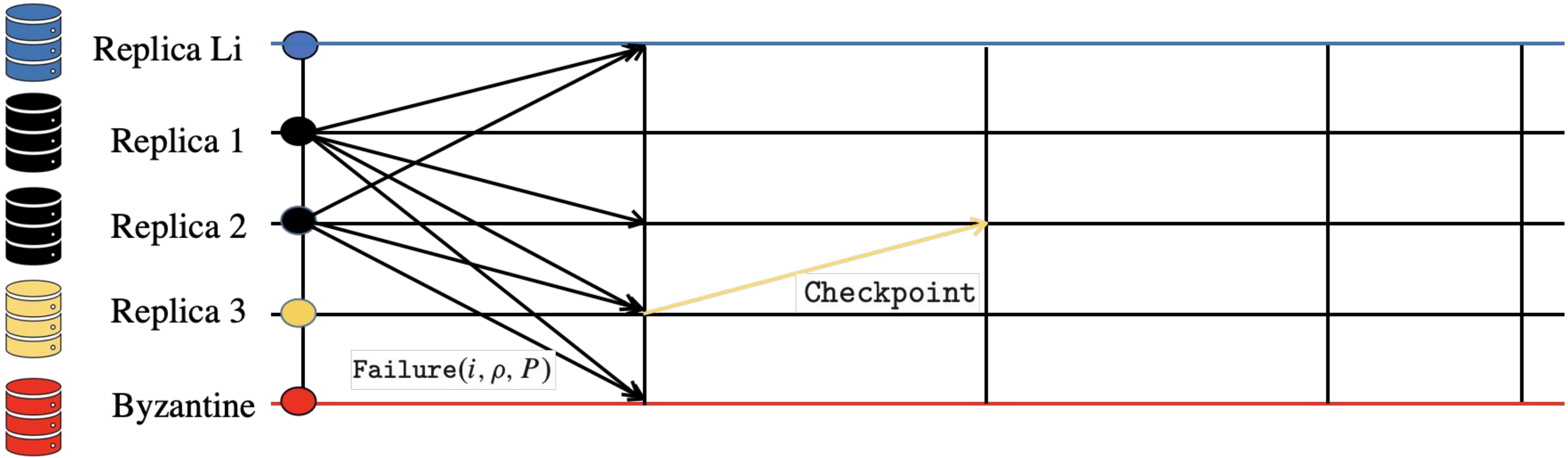
$n = 3f + 1 = 7$

In-the-dark attacks

Dealing with Undetectable Failures

- A standard checkpoint algorithm
- On a dynamic per-need basis

Dealing with Undetectable Failures



Dealing with Failures

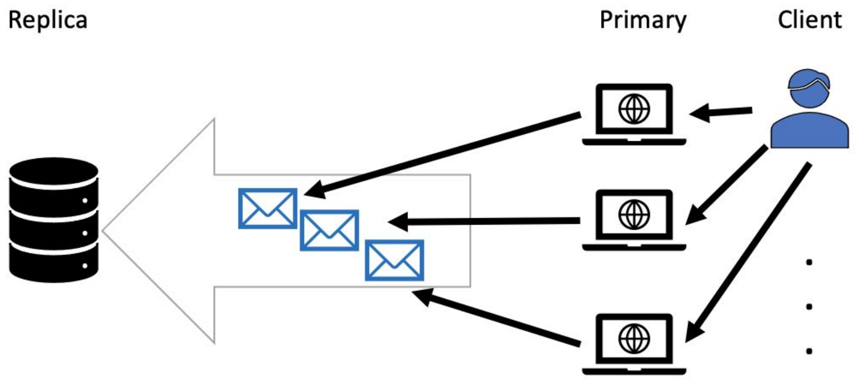
Theorem: *Consider RCC running in a system with n replicas. If $n > 3f$, then RCC provides consensus in periods in which communication is reliable.*

Client Interactions with RCC

- Why is Client Interaction important?
- RCC's optimization
- Problem introduced by optimization
- Solution to the problems

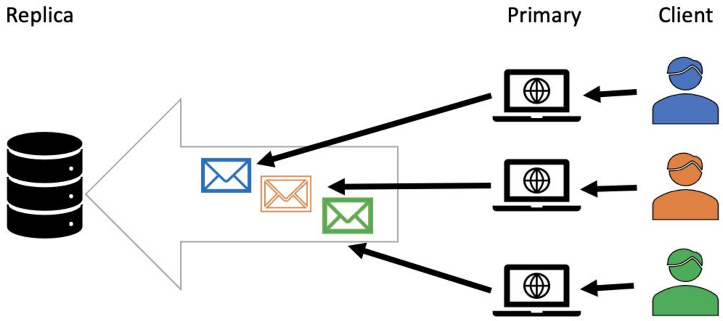
Why is Client Interaction important?

- RCC has multiple clients
- Proposing the same client transaction multiple times reduce throughput



RCC's Optimization

- Assign every client c_i to a single primary P_i such that only the primary P_i can propose c_i's requests
- Benefit: no need to coordinate clients to assure that they send their transactions to only a single primary

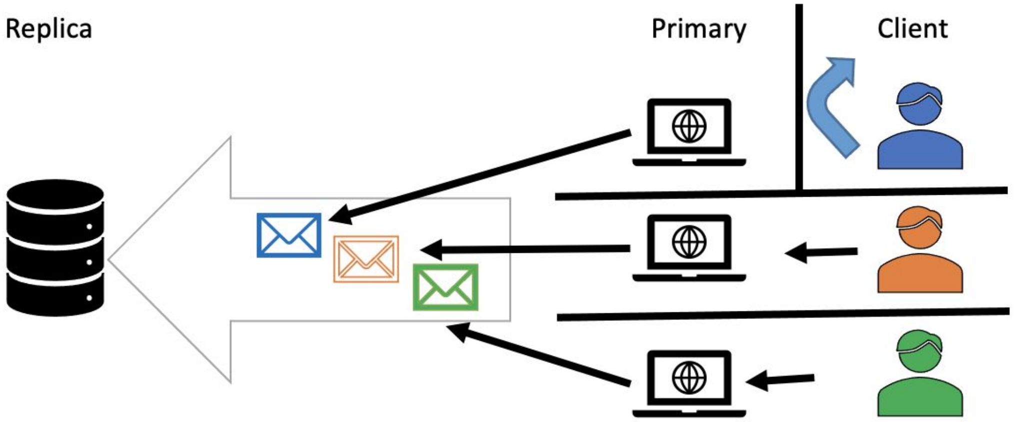


Problems introduced by optimization

- What if primaries do not receive client requests?

  Primary proposes a small no-op-request

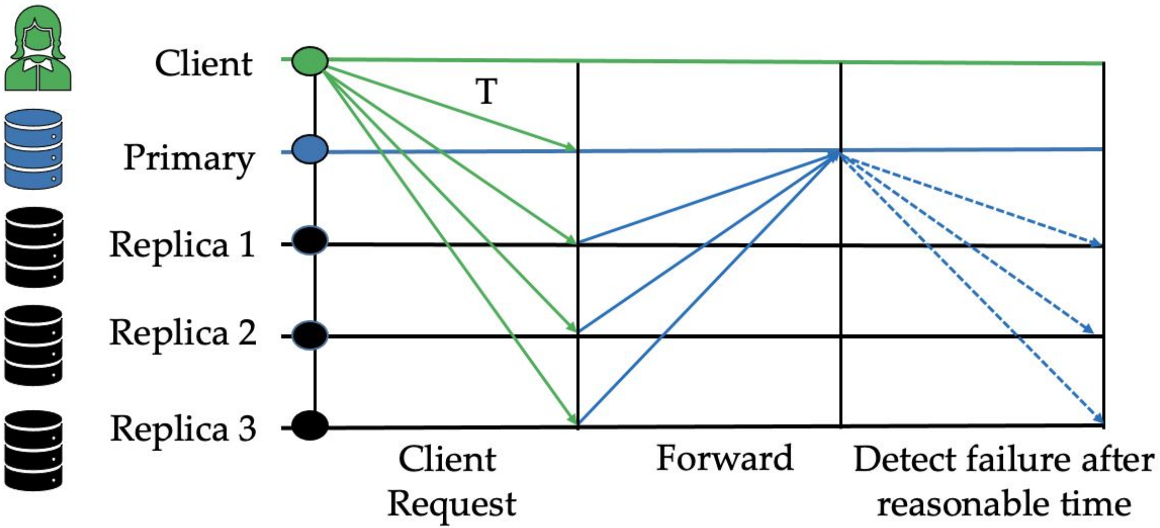
- What if faulty primaries refuse to propose requests of some clients?



Problems introduced by optimization

- What if faulty primaries refuse to propose requests of some clients?

Incentivize the fault primary to not refuse
Request reassignment



Client Interaction Summary

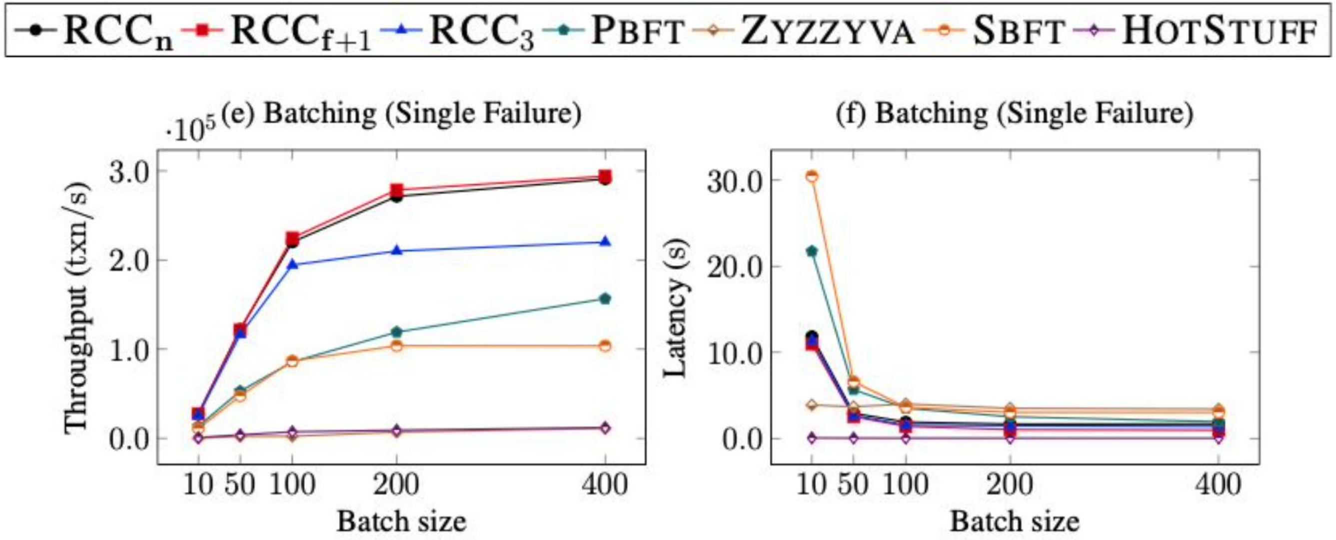
- Each client is assigned to a single primary
- What if primaries do not receive client requests?
  - Primary proposes a small no-op-request instead
- What if faulty primaries refuse to propose requests of some clients?
  - Malicious P: Incentivize malicious primaries to not refuse services
  - Crashed P: reassign primary

Evaluation of RCC

- Varying batch sizes
- Varying numbers of replicas
- Out-of-processing disabled
- Varying BFT protocols

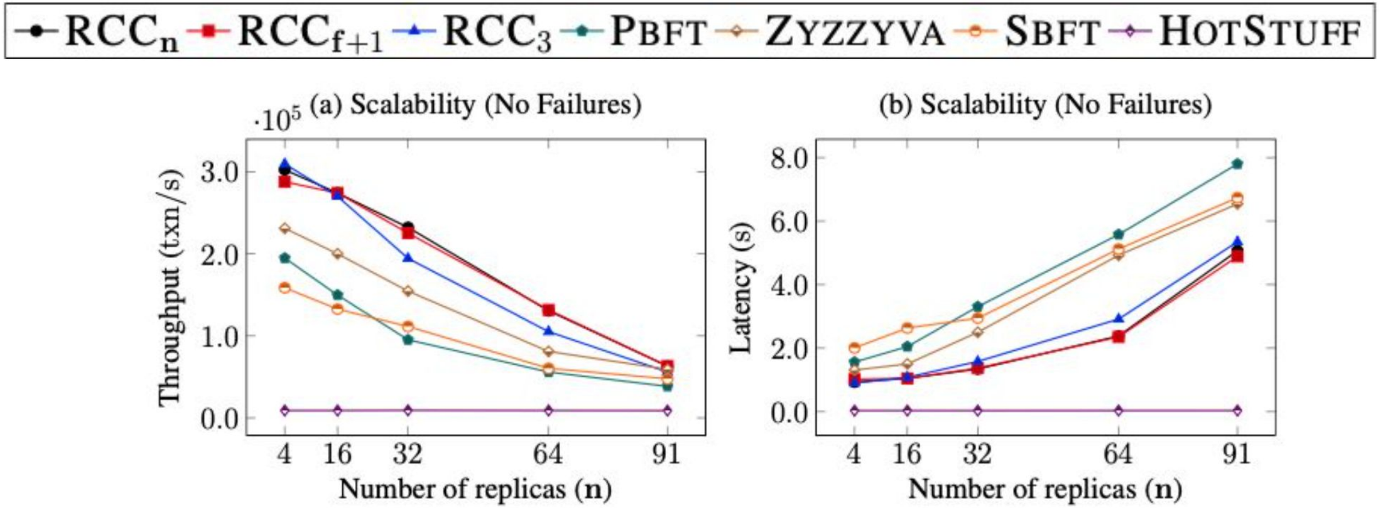
Evaluation - Varying Batch Size

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



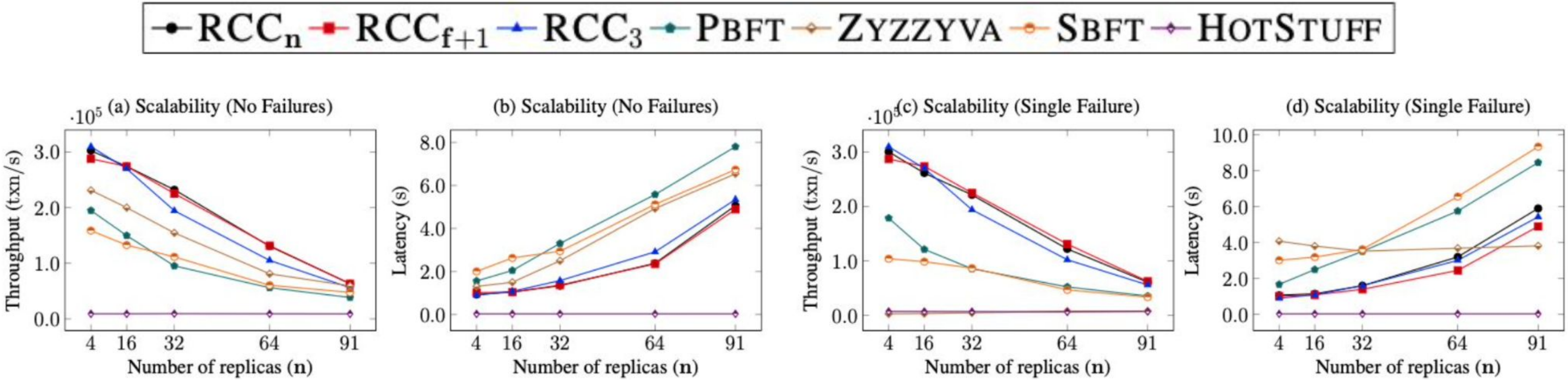
Evaluation - Varying #Replicas (No Failures)

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



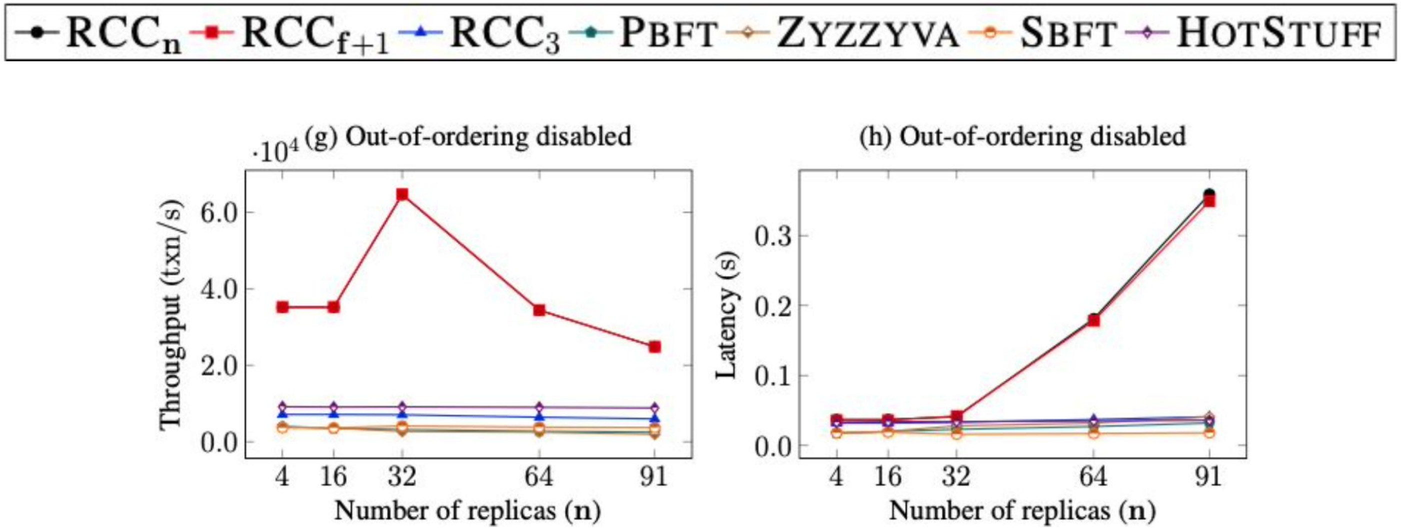
Evaluation - Varying #Replicas

- Three versions of RCC outperform all other protocols
- Performance of RCC with or without failures is comparable
- Adding concurrency by adding more instances improves performance
- On small deployments with n = 4, . . . , 16 replicas, the strength of RCC is most evident



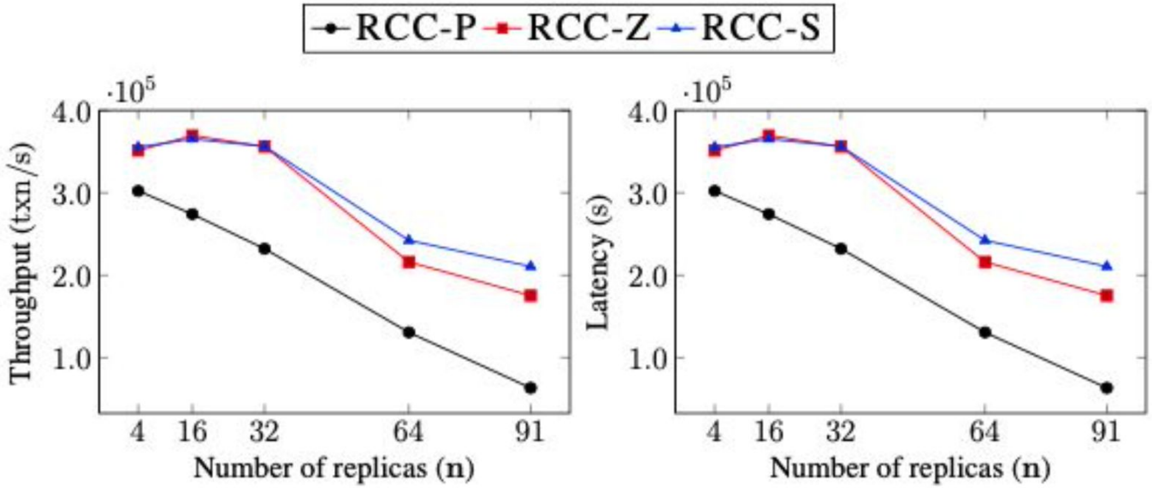
Evaluation - Out-of-ordering disabled

- HOTSTUFF 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



Evaluation - Varying BFT Protocol

- RCC-S and RCC-Z achieve higher throughput than RCC-P
- RCC-S consistently attains equal or higher throughput than RCC-Z



Evaluation - Conclusion

- RCC provides more throughput than any primary-backup consensus protocol can provide.
- RCC provides great scalability if throughput is only bounded by the primaries.
- RCC can efficiently deal with failures

Throughput	RCC / SBFT	RCC / PBFT	RCC / HotStuff	RCC / Zyzzyva
Single Failure	2.77x	1.53x	38x	82x
No Failure	2x	1.83x	33x	1.45x

Thank you

Appendix: Ordering (cont.)

Ordering attack example: we have transfer(A, B, m), A has \$5, B has \$0, C has \$0

- T1 from C1: transfer(A, B, \$3)
- T2 from C2: transfer(B, C, \$1)
- If primary decide T1 -> T2, then A:\$2, B:\$2, C:\$1
- But if primary decide T2->T1, then A:\$2, B:\$3, C:\$0
- Previously, the only primary can decide the order
- In RCC, all primary nodes decide the deterministic order together, so that no single primary can benefit themselves by deciding the order.

Appendix: Ordering

You can find more details in section: **IV. RCC: IMPROVING RESILIENCE OF CONSENSUS**.

RCC propose a method to deterministically select a different permutation of the order of execution in every round. In such a way that this ordering is partially impossible to predict or influence by faulty replicas.

There is a function that maps an integer h to a unique permutation(if we find a h, we find a permutation, if we find a permutation, we find an order). For sequence S of $k = |S|$ values, there exists k! distinct permutations, so each integer h from {0, ..., k!-1} will correspond to a permutation. Then pick **h = hash(accepted T)mod(k!-1)** each round to select that permutation order. Only the concurrent phase ends we know how many are accepted in this round.

Appendix: RCC & GeoBFT

- Similarities:
  - Runs multiple BCAs in parallel
  - Can apply to any other BCAs
  - Aims to increase throughput
  - After BCA, needs to order transactions before execution
- Differences:
  - Different concurrent mechanisms: Physically needs more replicas to run more BCAs for GeoBFT, while RCC logically runs more BCAs with the same amount of replicas
  - Different objectives: GeoBFT aims to resolve latency limitations, RCC aims to make full use of non-primary nodes
  - Different failures dealing mechanisms

References

[1] S. Gupta, J. Hellings, and M. Sadoghi. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering*. IEEE, 2021.

[2] S. Gupta, J. Hellings, and M. Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC)*, 146:44:1–44:3, Schloss Dagstuhl, 2019.

[3] S. Gupta, J. Hellings, and M. Sadoghi. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction https://www.youtube.com/watch?v=l15M1jyTyvo