Large Scale Systems
CS 410 / 510

Lecture 14:

Parallelizing Consensus

Suyash Gupta

Assistant Professor

Distopia Labs and ONRG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Assignment 4 is Out!

- Assignment 4 is due on **June 2, 2025** at **11:59pm PST**.
- Please start working with your groups.

Presentations

- Presentation slots are out!
- Each group will present their MiniSpanner in their respective time slots.
- Presentation Format:
  - Each group will get **15-16 min** to present their progress.
  - **5 min** for a working demo
  - **10 min** for Q/A.
  - Every student should present for around 4 minutes.
  - Every student should state their contributions.
  - Based on Q/A and presentation quality, grades will be decided for each student.

Reading Material

- For this class and next lecture:
  - Read **Chapters 3-4** from **Fault-Tolerant Distributed Transactions on Blockchain**.

Last Class

- Last class we looked at:

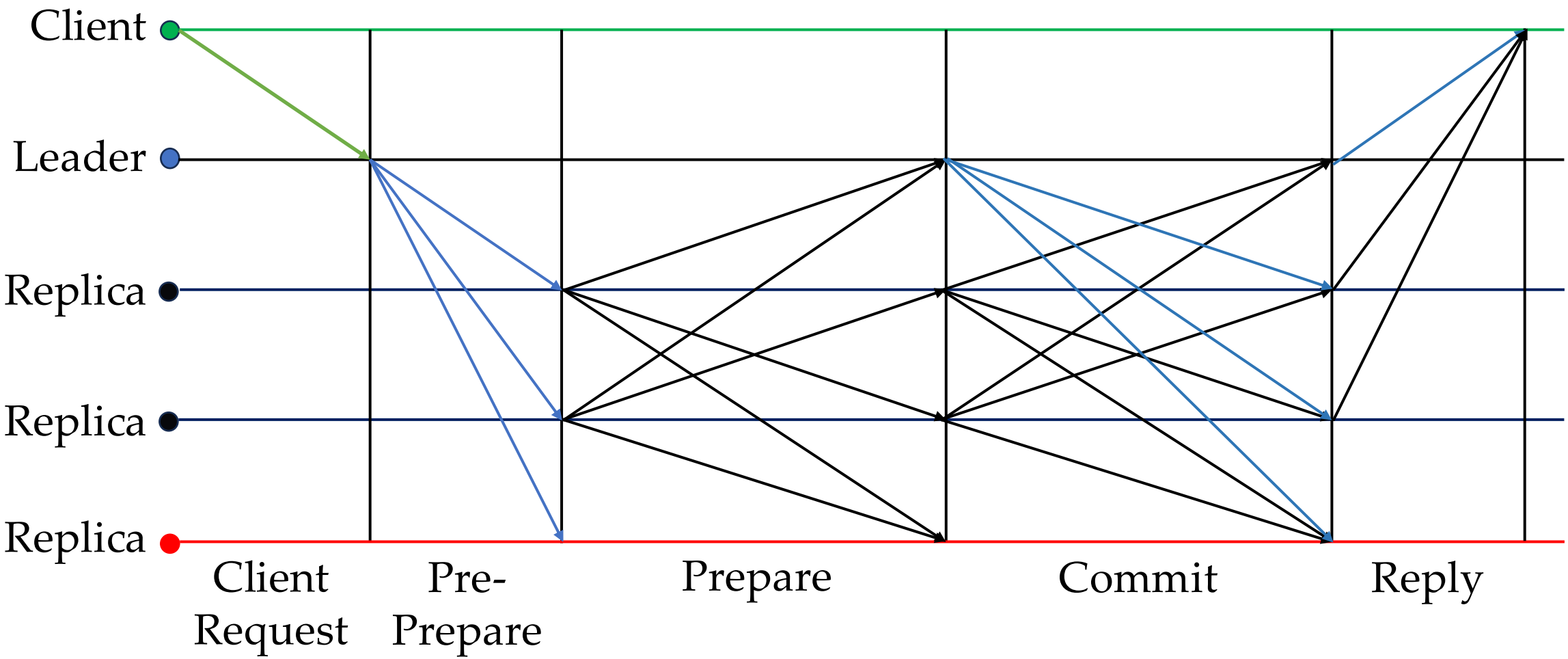
- Speculation in Consensus
- PoE Protocol

Optimizing PBFT

- Today, our goal is to optimize PBFT through parallelism.

- How can we introduce parallelism in PBFT apart from out-of-order message processing?

PBFT Protocol



Optimizing PBFT

- Today, our goal is to optimize PBFT through parallelism.

- How can we introduce parallelism in PBFT apart from out-of-order message processing?

Parallelizing PBFT

- What if we allow multiple replicas to act as the leader at the same time?
  - What if we allow all replicas to be the leader!

Parallelizing PBFT

- What if we allow multiple replicas to act as the leader at the same time?
  - What if we allow all replicas to be the leader!

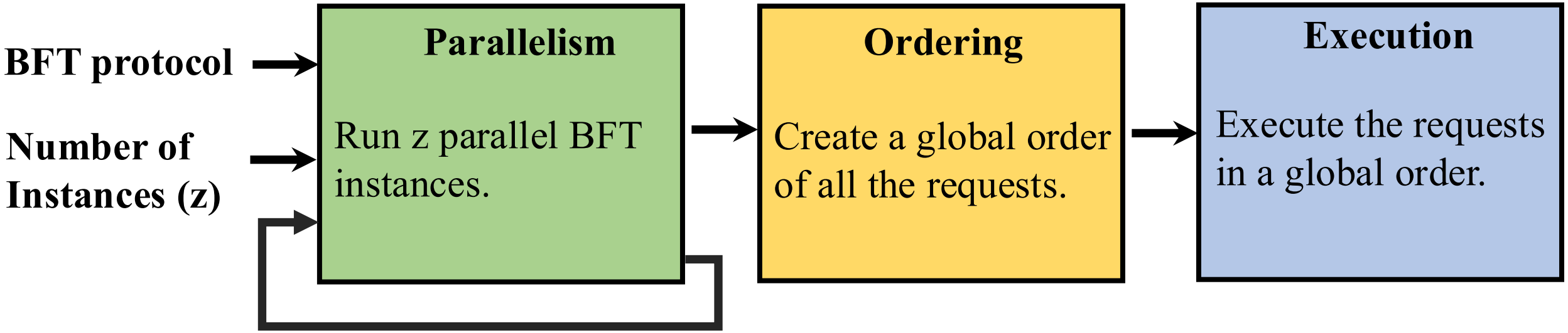
- To allow all replicas to act as leaders at the same time, we need to allow them to propose client requests at the same time → run consensus at the same time.

- Essentially, we are looking at a protocol where multiple instances of PBFT protocol are running in parallel.

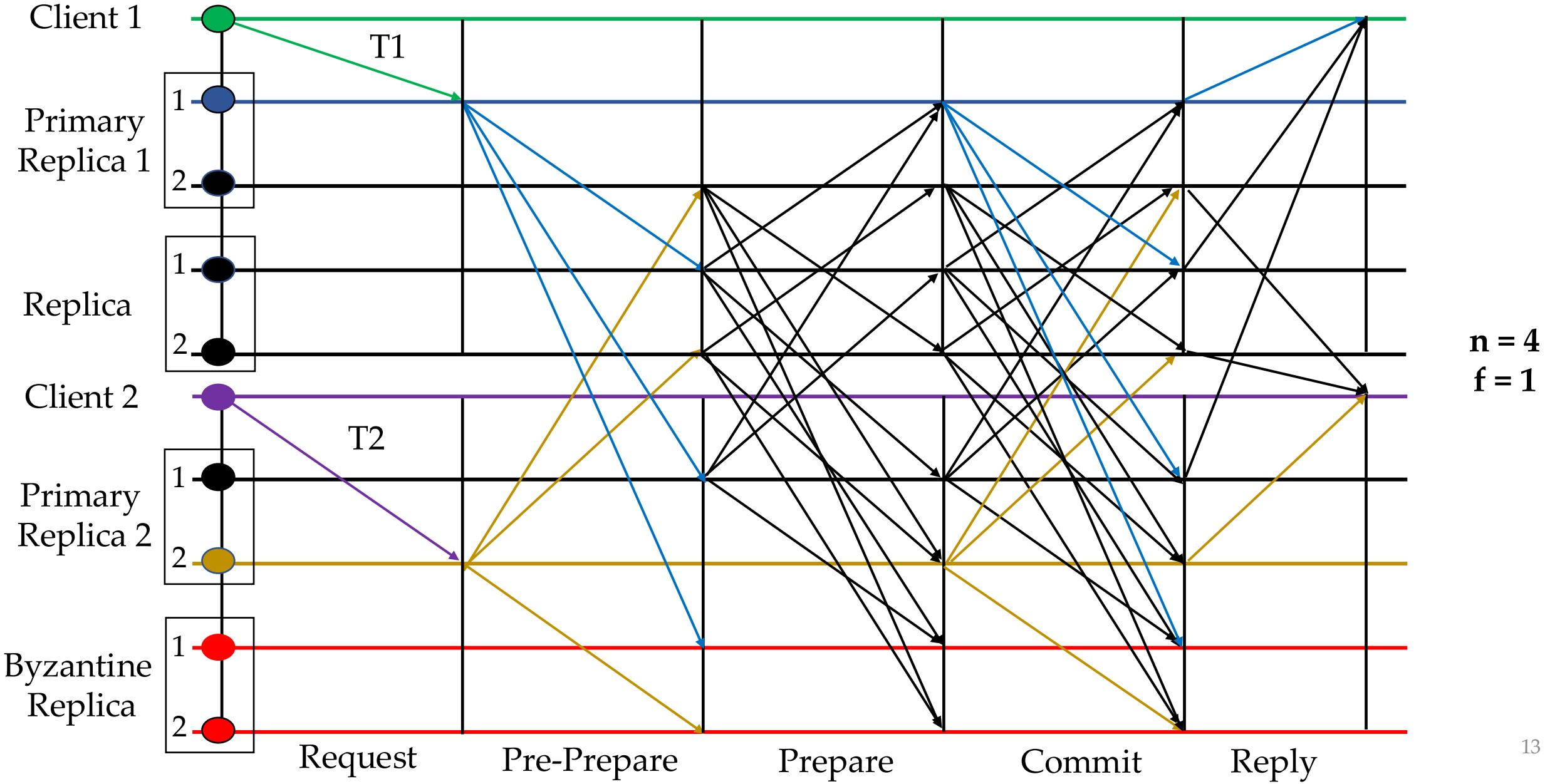
Resilient Concurrent Consensus

- **RCC [ICDE'21]** allows running multiple instances of PBFT in parallel.
- Either all replicas can act as leaders or a subset of replicas act as leaders.
- Total replicas: **n = 3f + 1**
  - Maximum number of Byzantine Replicas: **f**

RCC Protocol Overview



RCC Protocol with 2 Leaders



Ordering and Execution in RCC

- In what order should we execute the requests from all instances?

Ordering and Execution in RCC

- In what order should we execute the requests from all instances?
  - We need a global order of all requests.

Ordering and Execution in RCC

- In what order should we execute the requests from all instances?
  - We need a global order of all requests.
  - **Simplest solution**:
    - Create a deterministic order.
    - Pre-assign each instance a unique identifier (e.g. 0,1,2,…,z).
    - Execute all transactions in the ascending/descending order of identifiers.

Out-of-Order + RCC

- Say, we enable out-of-order message processing in RCC, how do we order and execute requests now?

Out-of-Order + RCC

- Say, we enable out-of-order message processing in RCC, how do we order and execute requests now?

- Recall that with out-of-order, transaction 2 can commit before transaction 1.

Out-of-Order + RCC

- Say, we enable out-of-order message processing in RCC, how do we order and execute requests now?

- Recall that with out-of-order, transaction 2 can commit before transaction 1.

- As all instances are working out-of-order, we assume existence of **epochs**.

Out-of-Order + RCC

- Say, we enable out-of-order message processing in RCC, how do we order and execute requests now?

- Recall that with out-of-order, transaction 2 can commit before transaction 1.

- As all instances are working out-of-order, we assume existence of **epochs**.
  - In every epoch, each instance is expected to finish one transaction.

  - E.g., at the end of first epoch → each instance should finish its first transaction.

Out-of-Order + RCC

- Say, we enable out-of-order message processing in RCC, how do we order and execute requests now?

- Recall that with out-of-order, transaction 2 can commit before transaction 1.

- As all instances are working out-of-order, we assume existence of **epochs**.
  - In every epoch, each instance is expected to finish one transaction.

  - E.g., at the end of first epoch → each instance should finish its first transaction.

  - But, epochs are forcing lock-step synchrony?

Out-of-Order + RCC

- Say, we enable out-of-order message processing in RCC, how do we order and execute requests now?

- Recall that with out-of-order, transaction 2 can commit before transaction 1.

- As all instances are working out-of-order, we assume existence of **epochs**.
  - In every epoch, each instance is expected to finish one transaction.
  - E.g., at the end of first epoch → each instance should finish its first transaction.
  - But, epochs are forcing lock-step synchrony?
  - **Allow consensus to keep proceeding but halt execution at epoch boundary**.

Key Challenges for RCC

- What are the key challenges for RCC?

Key Challenges for RCC

- What are the key challenges for RCC?
  - How to securely order the requests?

Key Challenges for RCC

- What are the key challenges for RCC?
  - How to securely order the requests?
  - Does failure of one instance causes other instances to stop?

Key Challenges for RCC

- What are the key challenges for RCC?
  - How to securely order the requests?
  - Does failure of one instance causes other instances to stop?
  - Can there be continuous ordering of requests?

Key Challenges for RCC

- What are the key challenges for RCC?
  - How to securely order the requests?
  - Does failure of one instance causes other instances to stop?
  - Can there be continuous ordering of requests?
  - How to defend against coordinated attacks?

Multi-Leader Dilemma

Multi-Leader Dilemma

- Having multiple leaders and multiple parallel instances can **boost throughput**!
- But now we must deal with **multiple Byzantine leaders** at the same time.
  - **PBFT** → only one leader → at most **one** Byzantine leader.
  - **RCC** → multiple leaders → at most **f** Byzantine leaders!

Need for Secure Ordering

Need for Secure Ordering

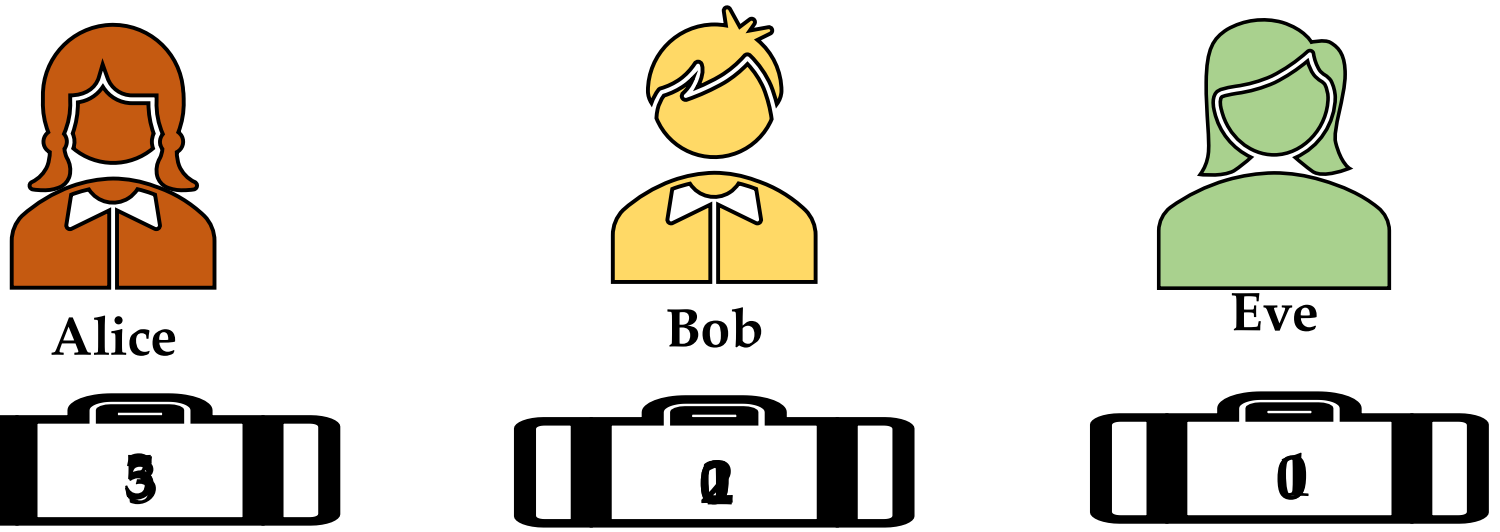
transfer(A,B,m) → *if*

bal(A) > m

then

bal(A) = bal(A) – m;
bal(B) = bal(B) + m;

Benefit to Bob!



T1: transfer(Alice, Bob, 2)

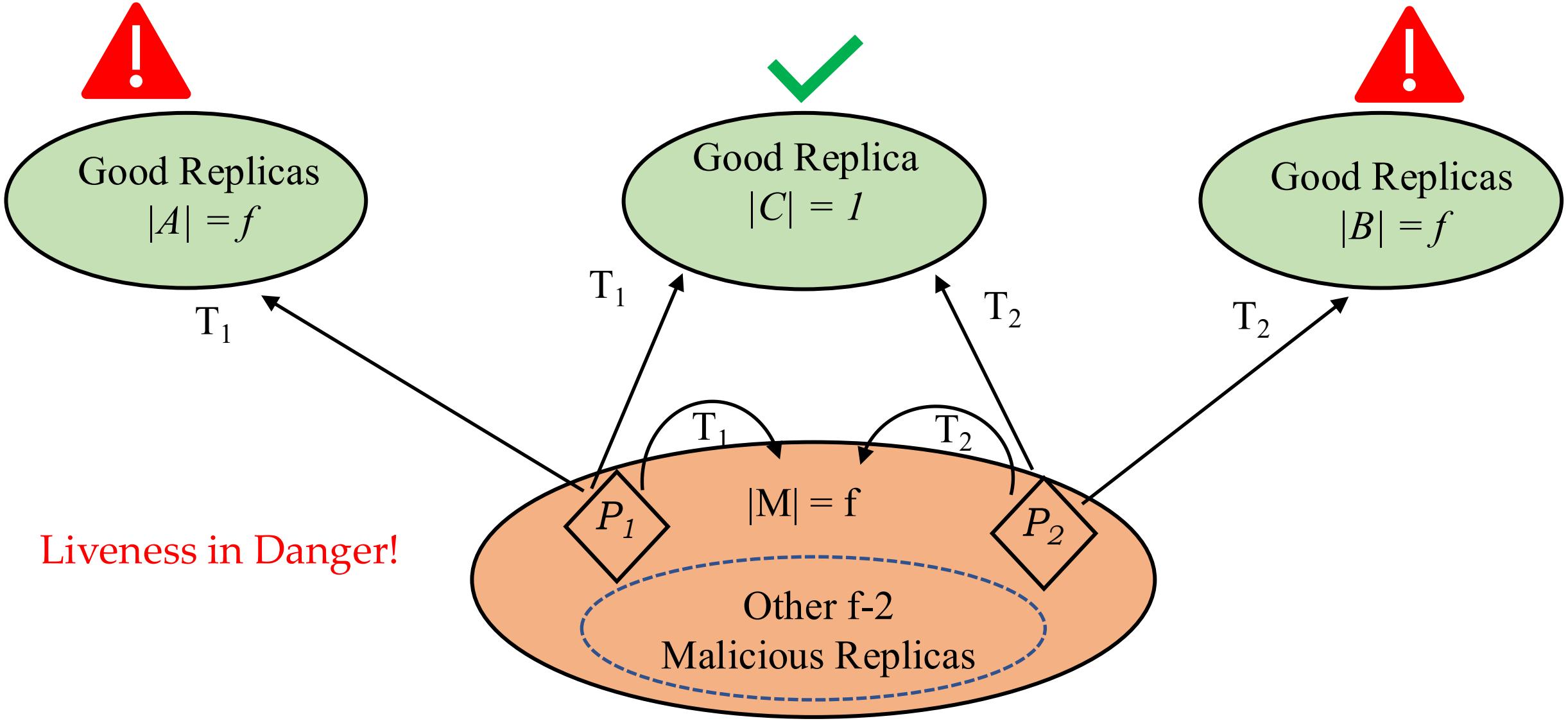
T2: transfer(Bob, Eve, 1)

Case 1: T1 then T2

Case 2: T2 then T1

Collusion by Malicious Primaries

Collusion by Malicious Primaries



Liveness in Danger!

Guaranteeing Secure Ordering

Guaranteeing Secure Ordering

- The goal for secure ordering is to ensure that the Byzantine parties cannot predict the order in which requests should get executed.

Guaranteeing Secure Ordering

- The goal for secure ordering is to ensure that the Byzantine parties cannot predict the order in which requests should get executed.

- **Solution:** Create a **random** permutation or sequence of requests.

Guaranteeing Secure Ordering

- The goal for secure ordering is to ensure that the Byzantine parties cannot predict the order in which requests should get executed.

- **Solution:** Create a **random** permutation or sequence of requests.

- **Protocol:**
  - At the end of each epoch, each replica collects all the committed requests.

Guaranteeing Secure Ordering

- The goal for secure ordering is to ensure that the Byzantine parties cannot predict the order in which requests should get executed.

- **Solution:** Create a **random** permutation or sequence of requests.

- **Protocol:**
  - At the end of each epoch, each replica collects all the committed requests.

  - Each replica creates a sequence of these requests.

Guaranteeing Secure Ordering

- The goal for secure ordering is to ensure that the Byzantine parties cannot predict the order in which requests should get executed.

- **Solution:** Create a **random** permutation or sequence of requests.

- **Protocol:**
  - At the end of each epoch, each replica collects all the committed requests.
  - Each replica creates a sequence of these requests.
  - Each replica hashes this sequence.

Guaranteeing Secure Ordering

- The goal for secure ordering is to ensure that the Byzantine parties cannot predict the order in which requests should get executed.

- **Solution:** Create a **random** permutation or sequence of requests.

- **Protocol:**
  - At the end of each epoch, each replica collects all the committed requests.
  - Each replica creates a sequence of these requests.
  - Each replica hashes this sequence.
  - Calls a function that randomly reorders the sequence based on a seed (hash).

Guaranteeing Secure Ordering

We define a function f_S recursively:

$$f_S(i) = \begin{cases} S & \text{if } |S| = 1; \\ f_{S \setminus S[q]}(r) \oplus S[q] & \text{if } |S| > 1, \end{cases}$$

Here,

$q = i \div (|S| - 1)!$

$r = i \bmod (|S| - 1)!$

Observe,

f_S partitions any value j, $0 \leq j \leq |S|! - 1$ into q and r.

Preventing Malicious Collusion

Preventing Malicious Collusion

- **Detection:**
  - To detect a collusion attack, we cannot rely on the same principles of detecting a malicious leader.

Preventing Malicious Collusion

- **Detection:**
  - To detect a collusion attack, we cannot rely on the same principles of detecting a malicious leader.
  - Detecting a malicious leader → If f+1 replicas complain.
  - Collusion Attack → No more than f replicas are complaining about the same leader.

Preventing Malicious Collusion

- **Detection:**
  - To detect a collusion attack, we cannot rely on the same principles of detecting a malicious leader.
  - Detecting a malicious leader → If f+1 replicas complain.
  - Collusion Attack → No more than f replicas are complaining about the same leader.
  - **Solution:**
    - If f+1 replicas complain about multiple leaders, assume a collusion attack.

Preventing Malicious Collusion

- **Resolution:**

Preventing Malicious Collusion

- **Resolution:**
  - Run the checkpoint protocol to exchange states.
  - But checkpoint protocol is expensive as full state needs to be exchanged.

Preventing Malicious Collusion

- **Resolution:**
  - Run the checkpoint protocol to exchange states.
  - But checkpoint protocol is expensive as full state needs to be exchanged.
  - **Solution:**
    - Run a lightweight checkpoint protocol.
    - Only exchange requests of the last epoch.