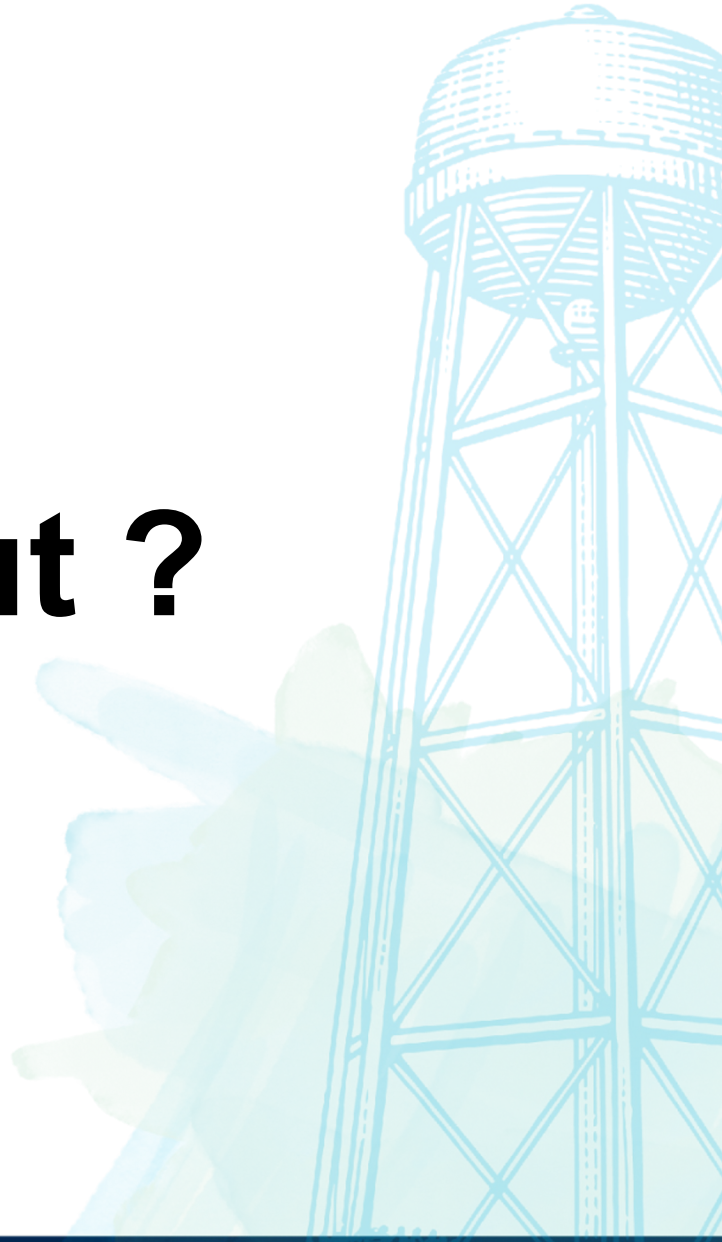
RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing

Authors: Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi

Presenters: Sankalp Kashyap, Rajaram Manohar Joshi, Shaik Haseeb Ur Rahman, Sanchit Kaul, Vijeth Kumbarahally Lakshminarayana

What is Throughput ?



Throughput

- Throughput refers to how many transactions (client requests) the system can process per second (txn/s).
- It's essentially a measure of the system's processing speed and efficiency.
- So, higher the throughput, higher the number of transactions per sec by the system.

Traditional Throughput Limitation

- Limited by single primary node's bandwidth since one node must send all transactions to everyone else.
- The theoretical max throughput (T_{max})

$$T_{max} = \frac{B}{(n-1)st}$$

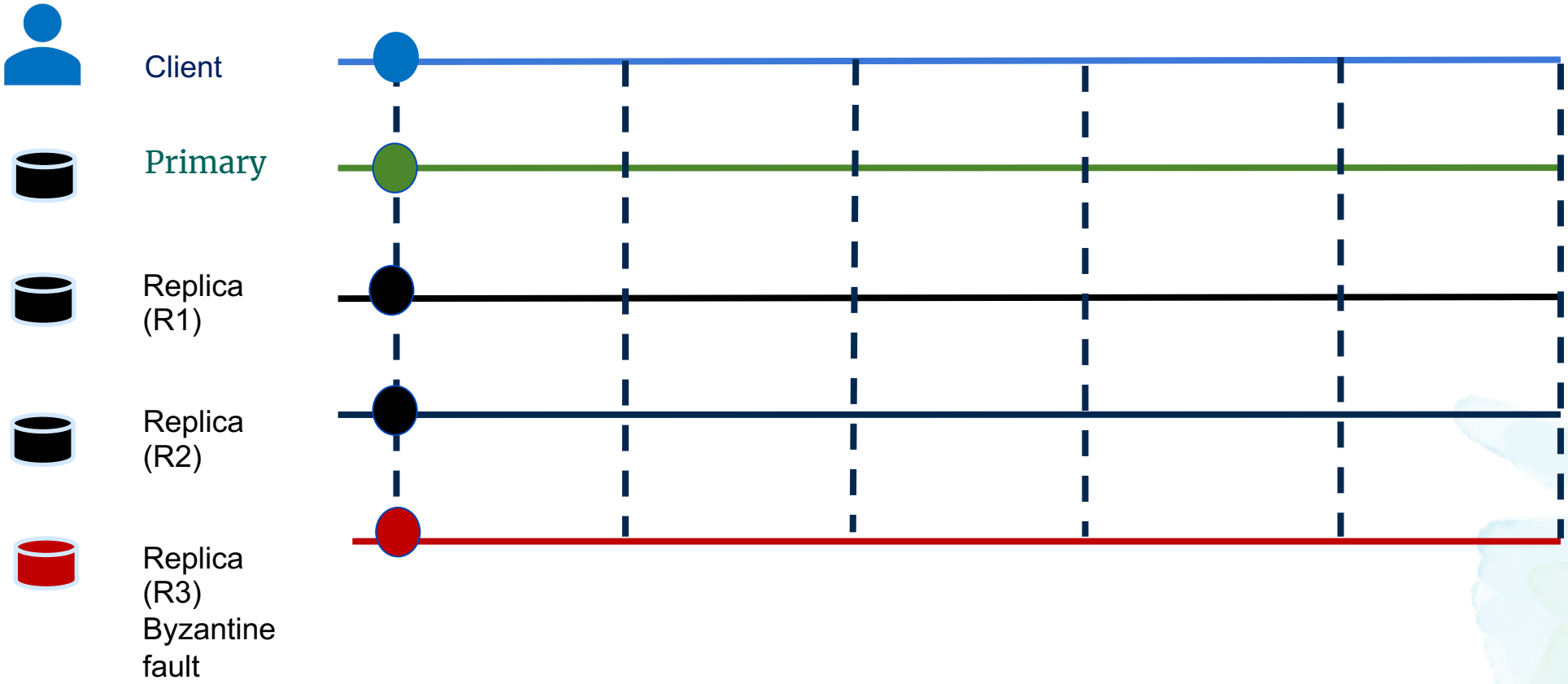
where, **B** is the outgoing Bandwidth of the primary

st is the size of each transaction.

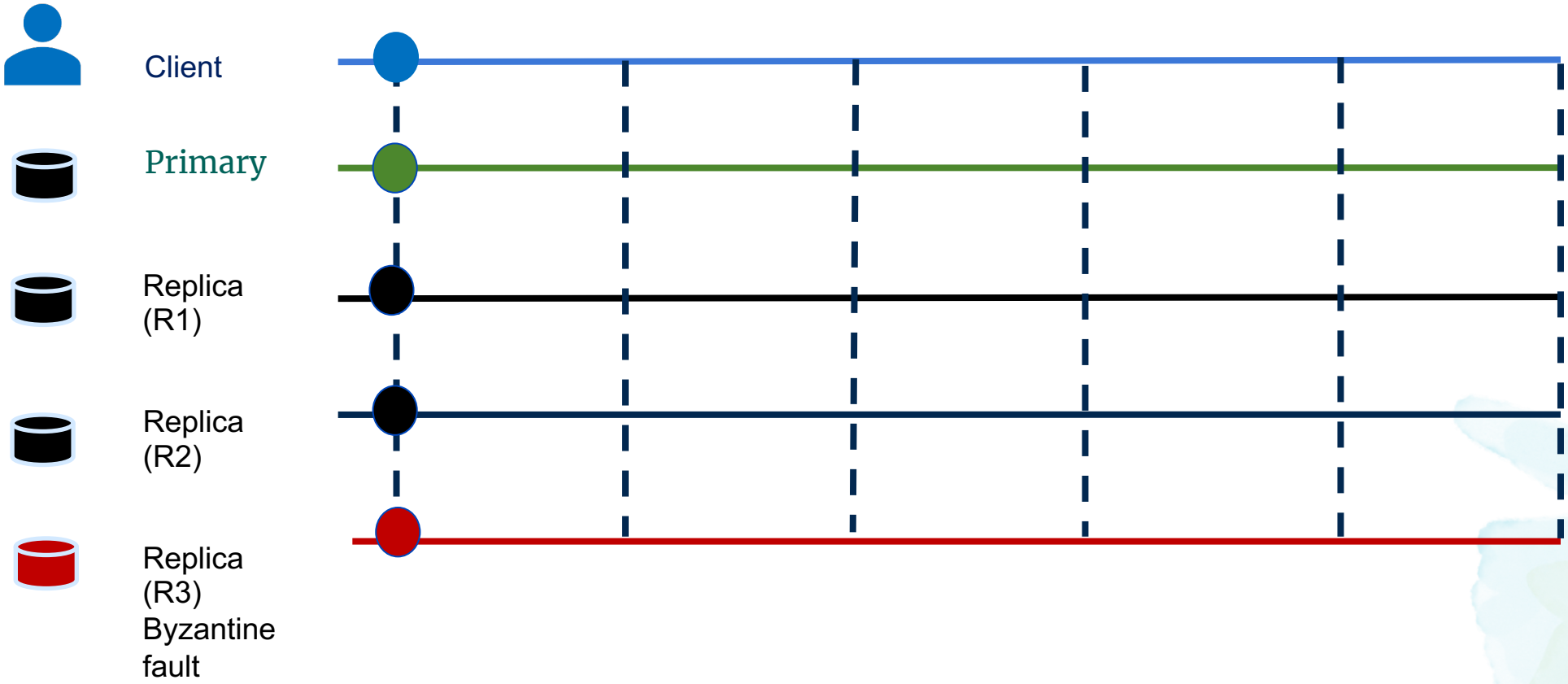
n is the total number of of nodes in the system

Throughput for PBFT

Overview of PBFT (Practical Byzantine Fault Tolerance)

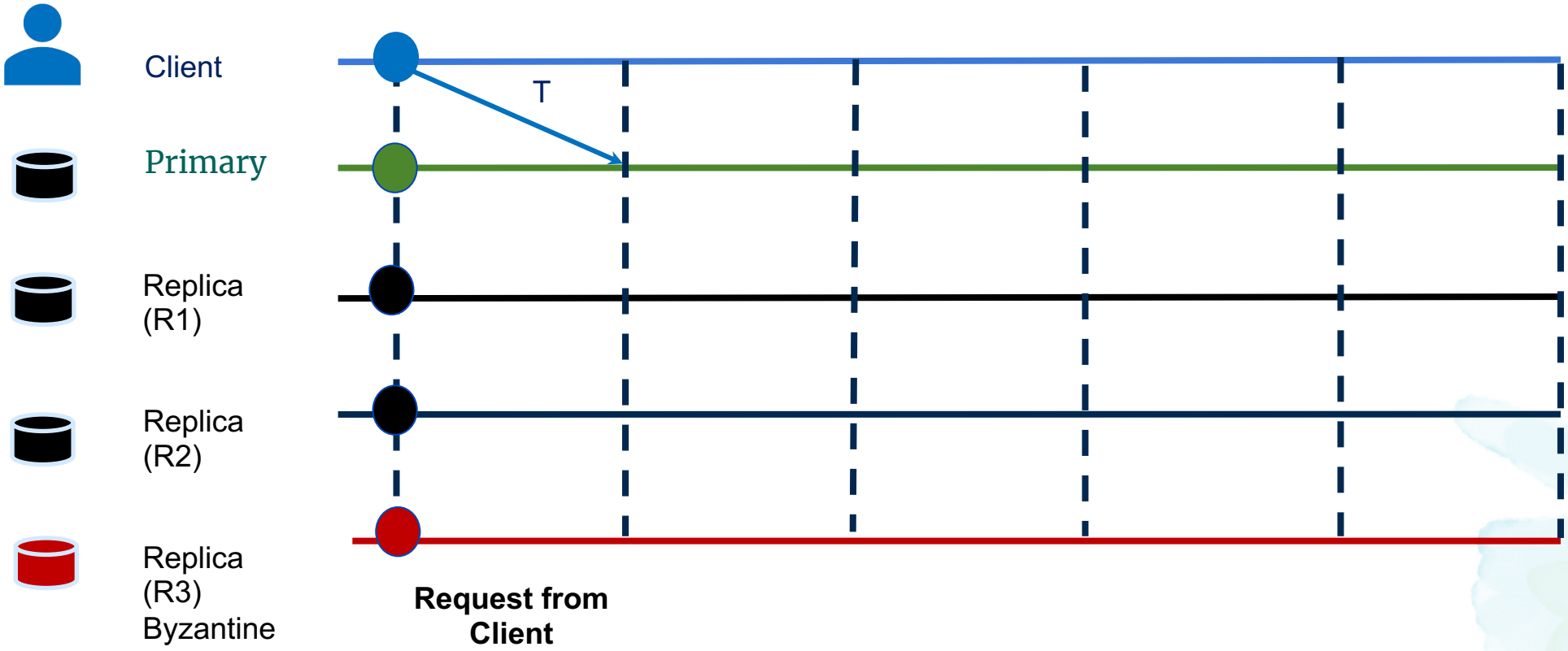


Overview of PBFT (Practical Byzantine Fault Tolerance)



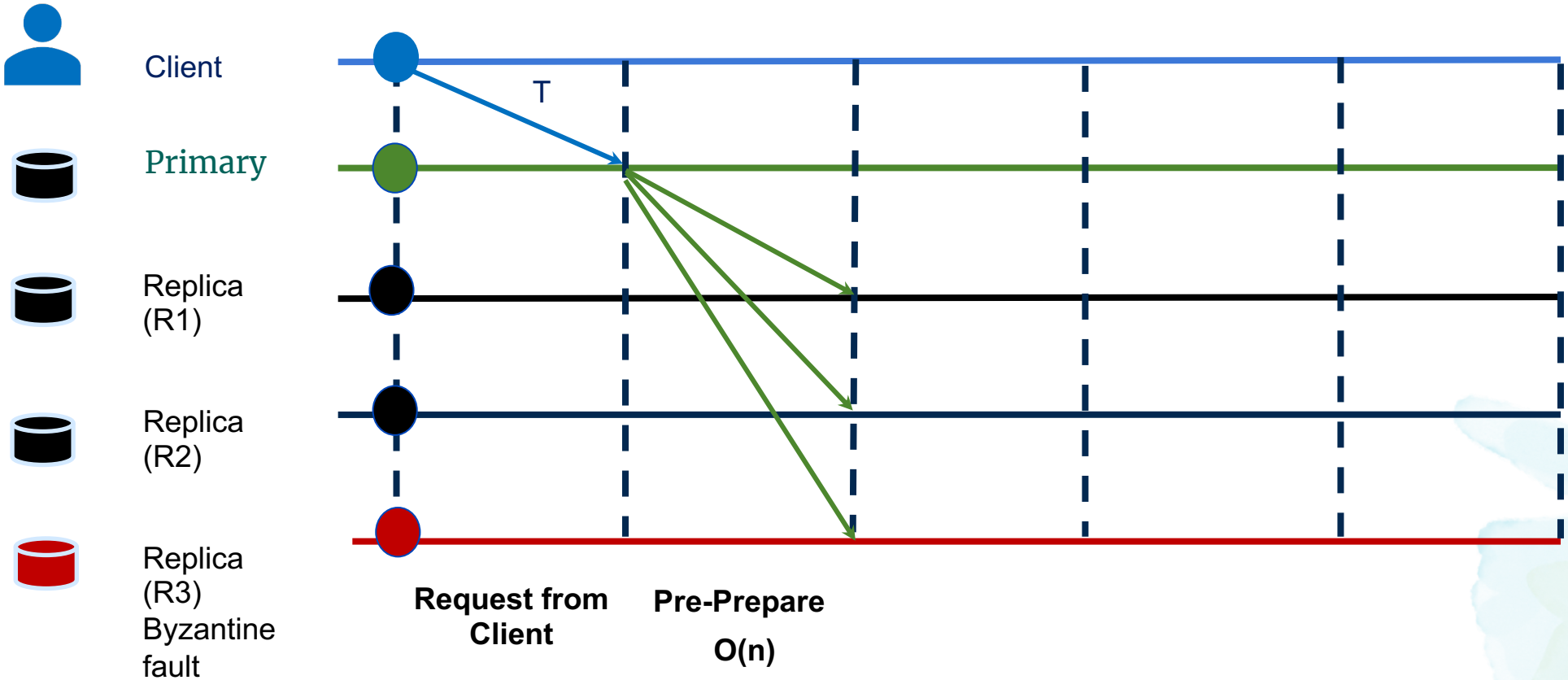
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



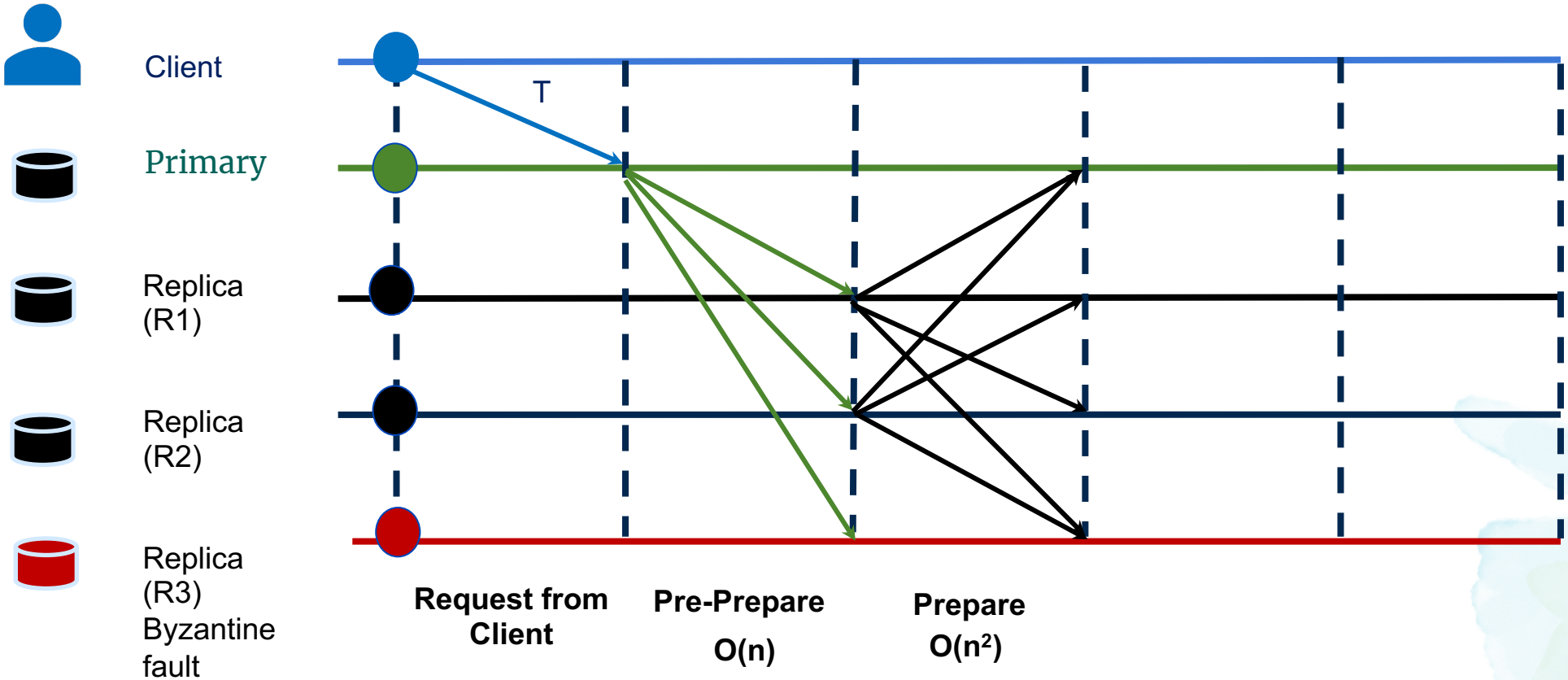
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



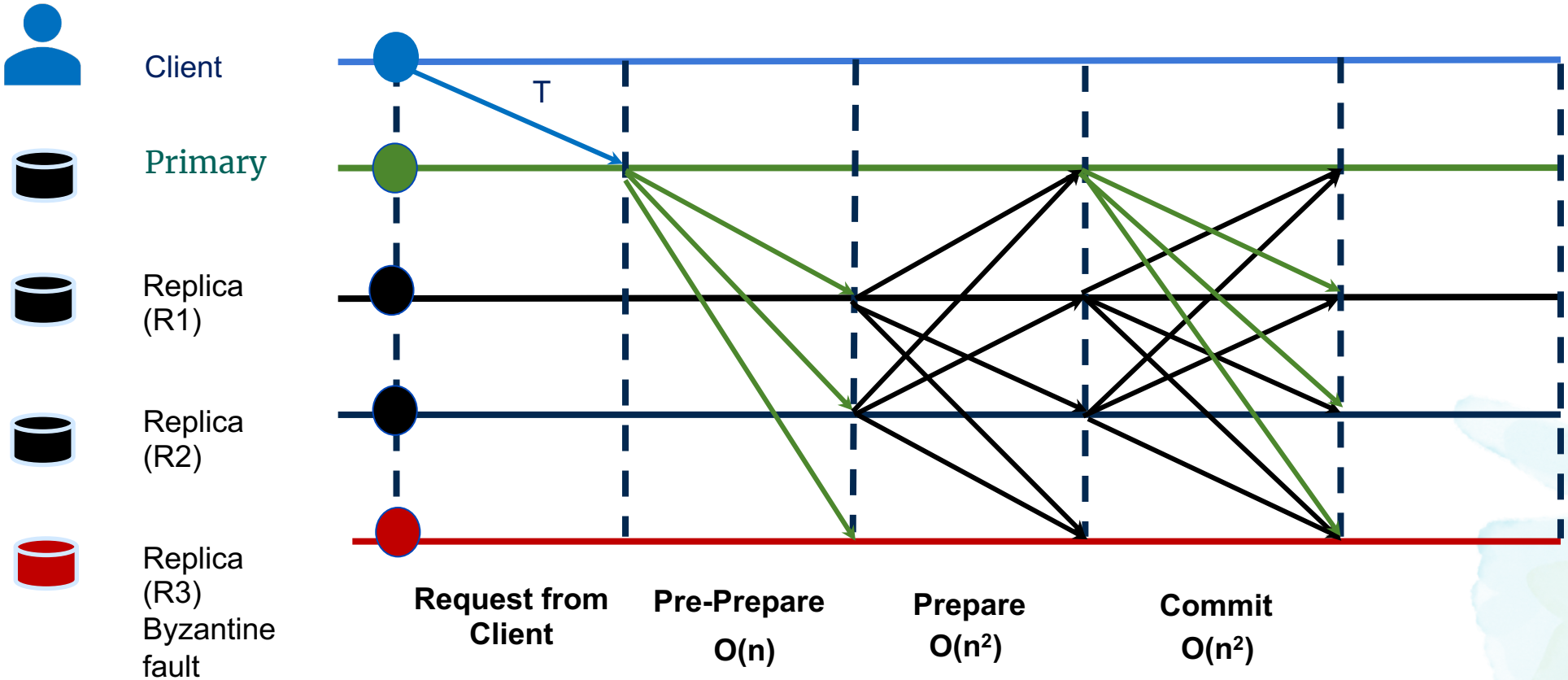
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



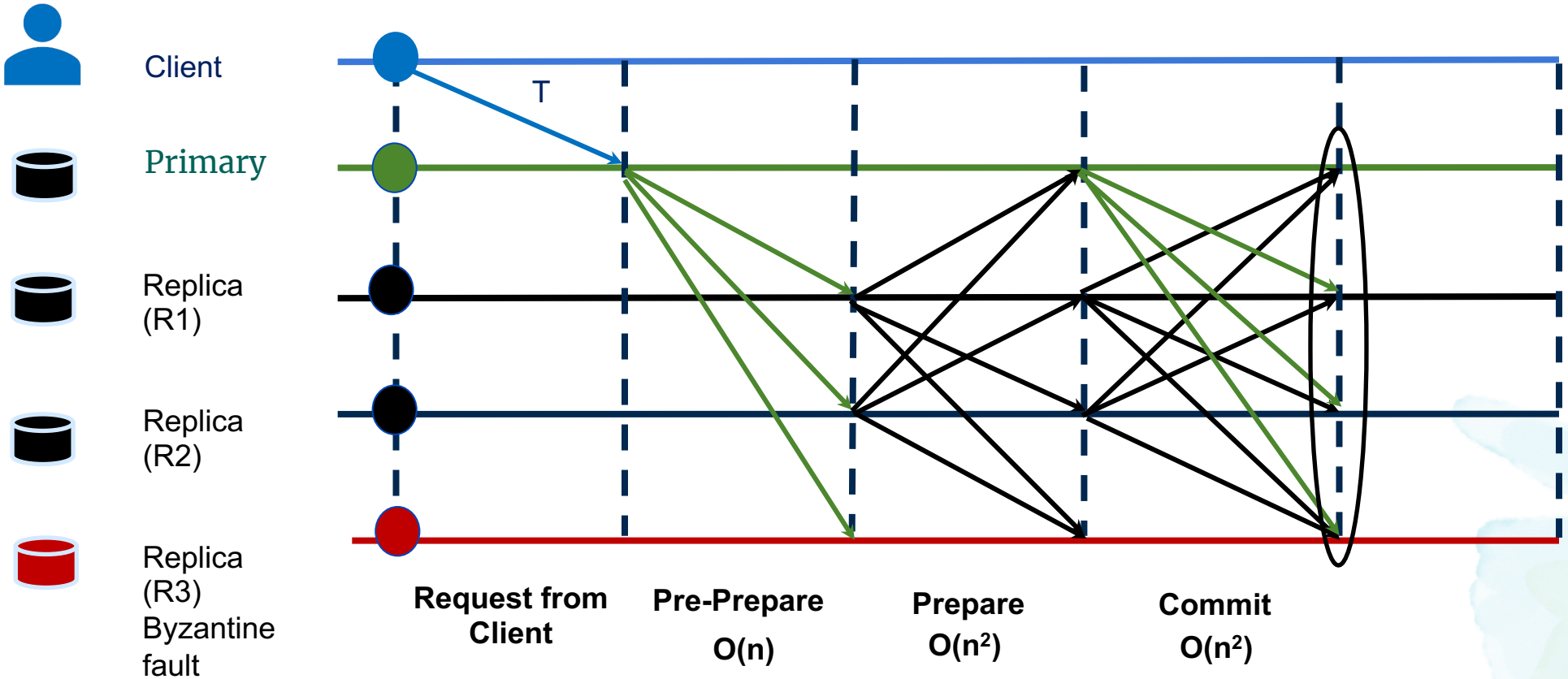
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



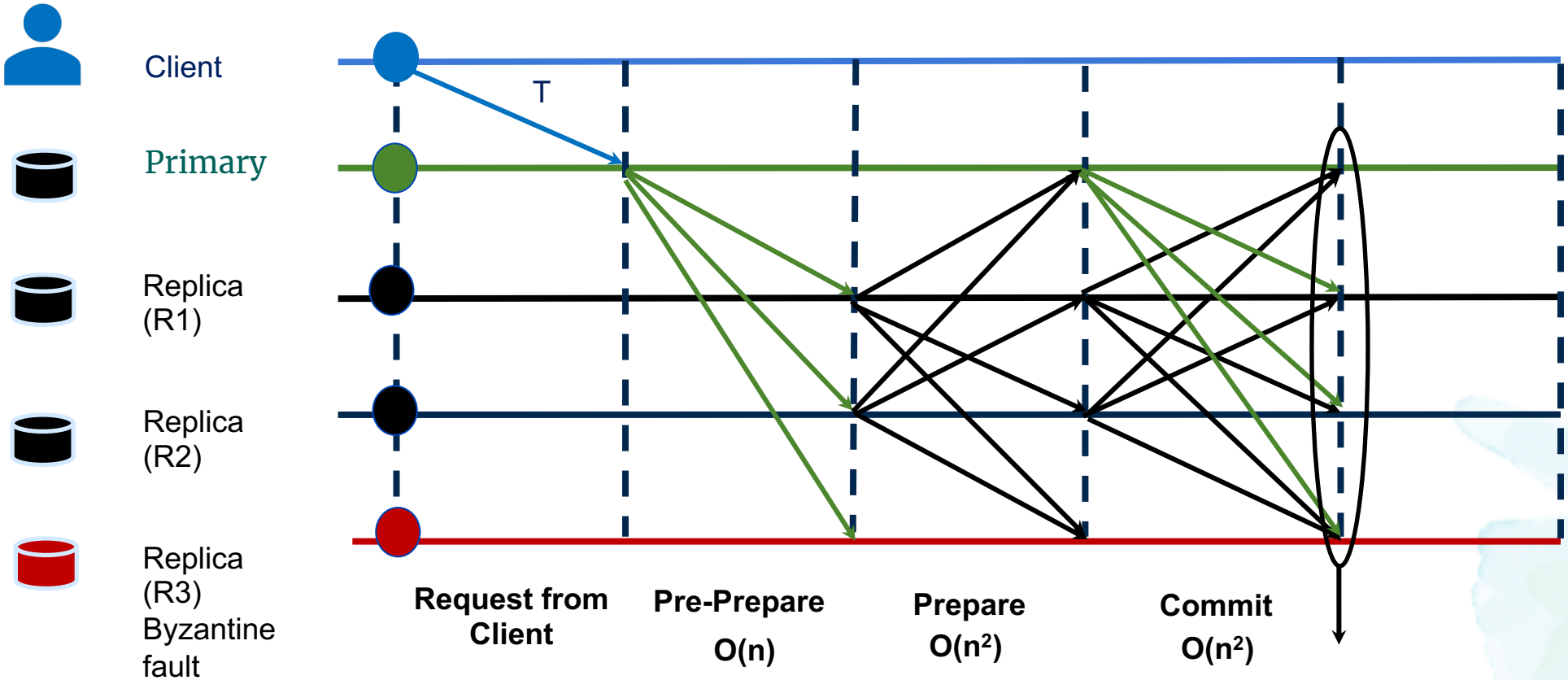
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



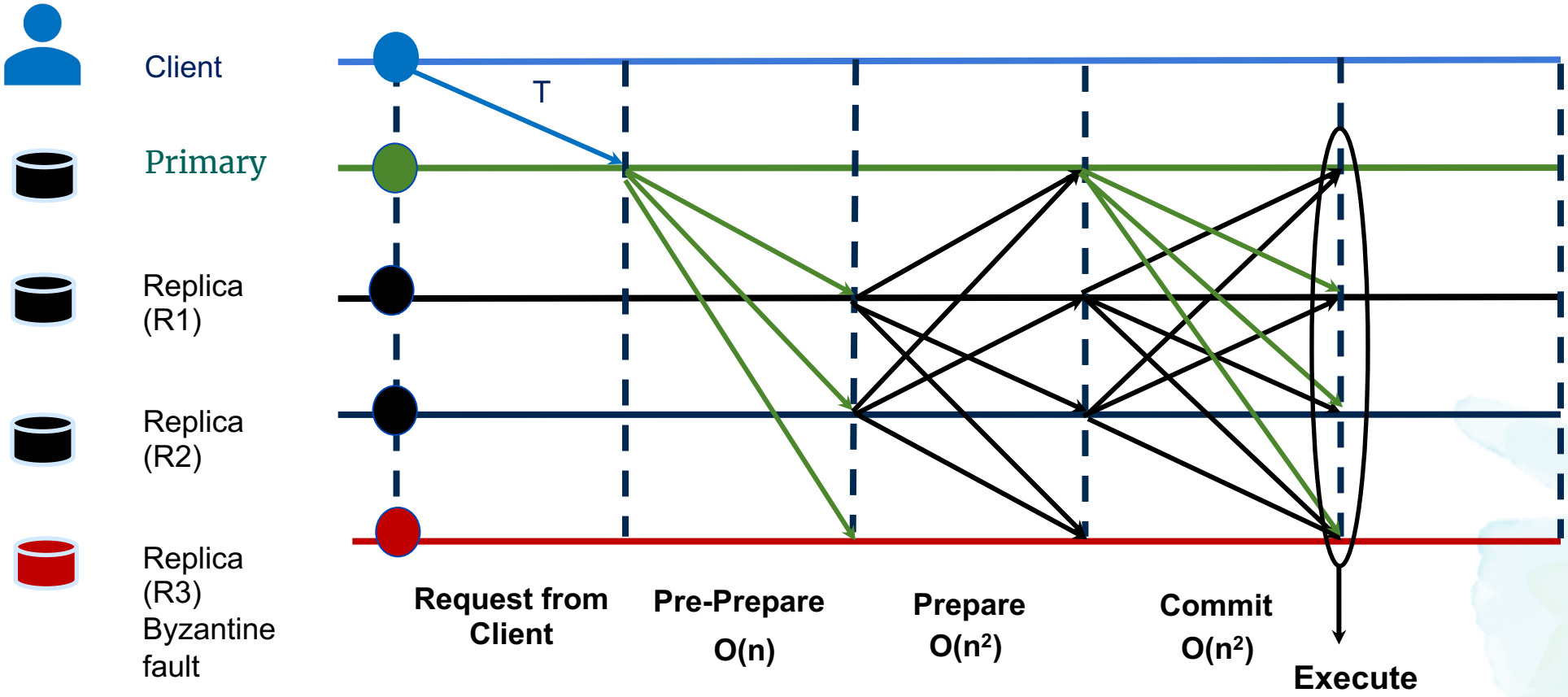
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



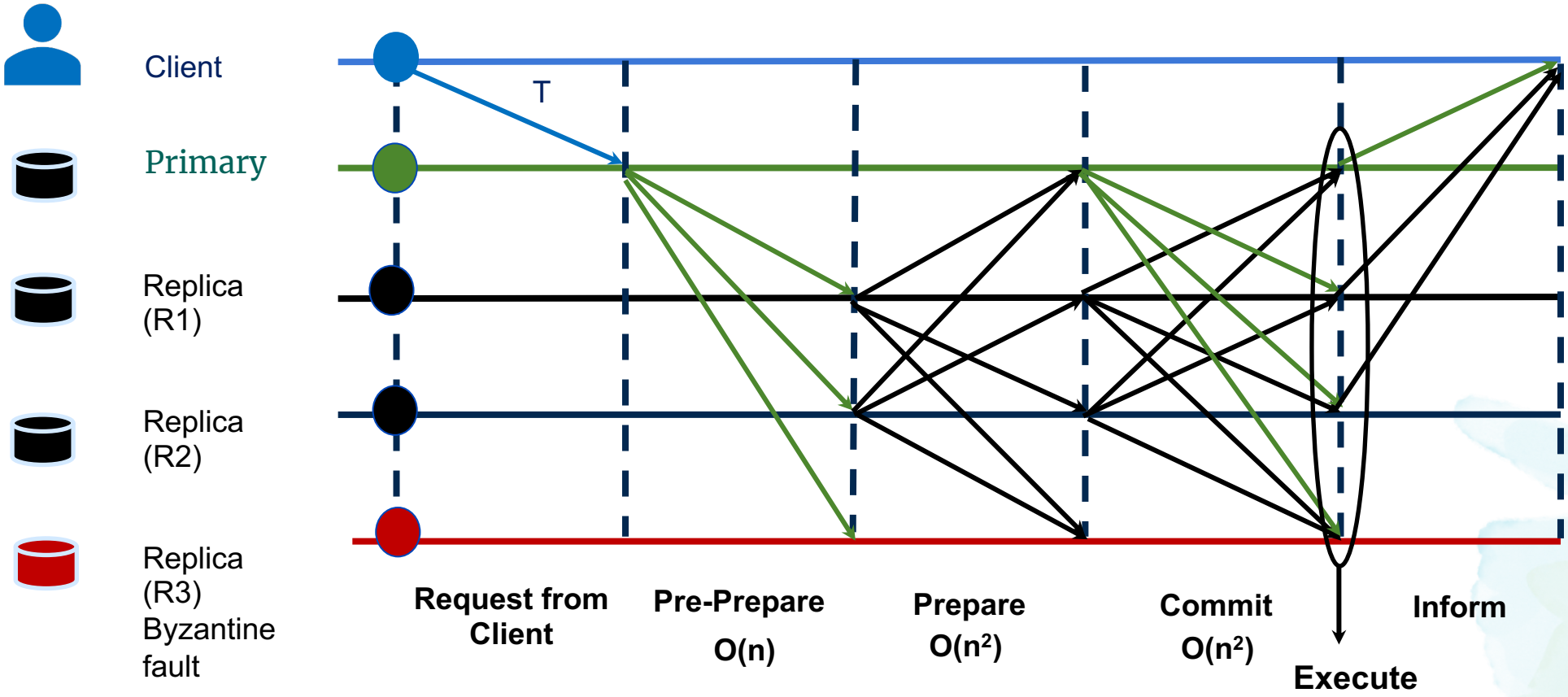
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



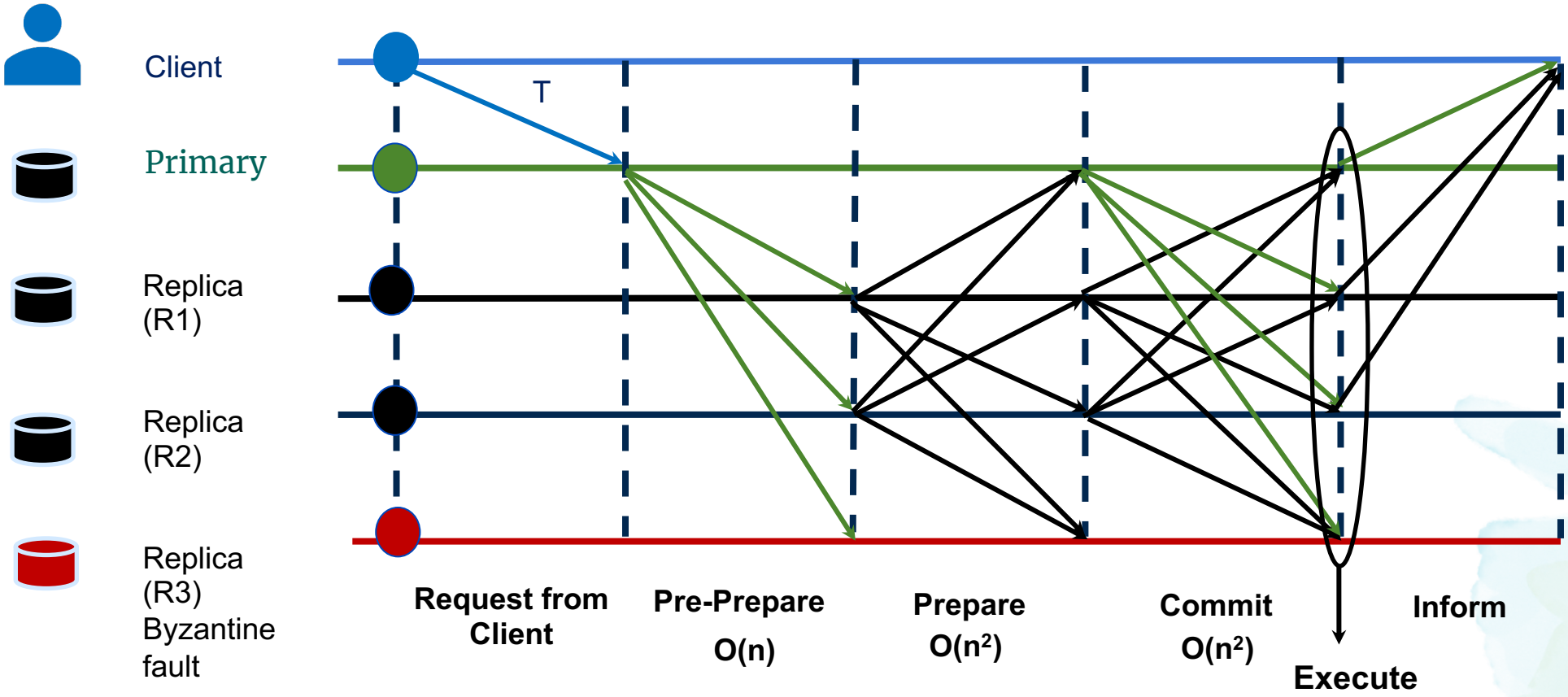
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



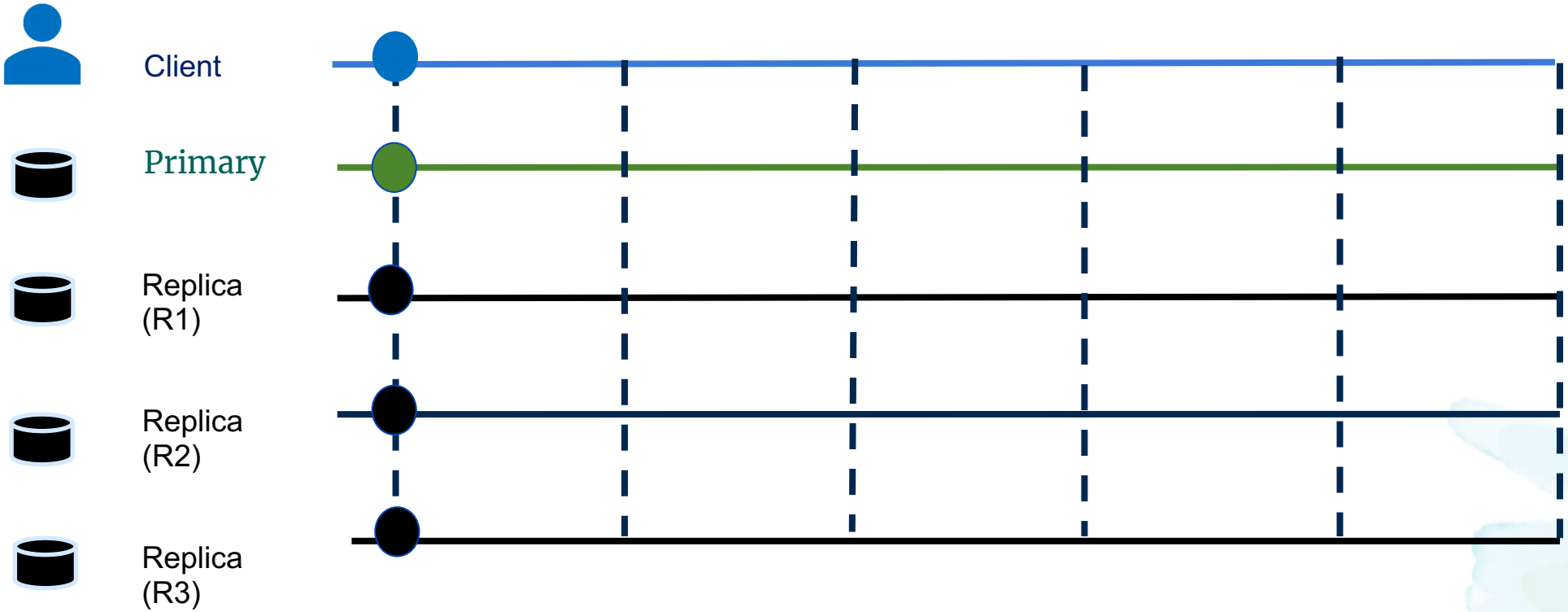
Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Overview of PBFT (Practical Byzantine Fault Tolerance)



Number of replicas (n) = 4 and Byzantine faults (f) <= 1

Understanding Throughput for PBFT

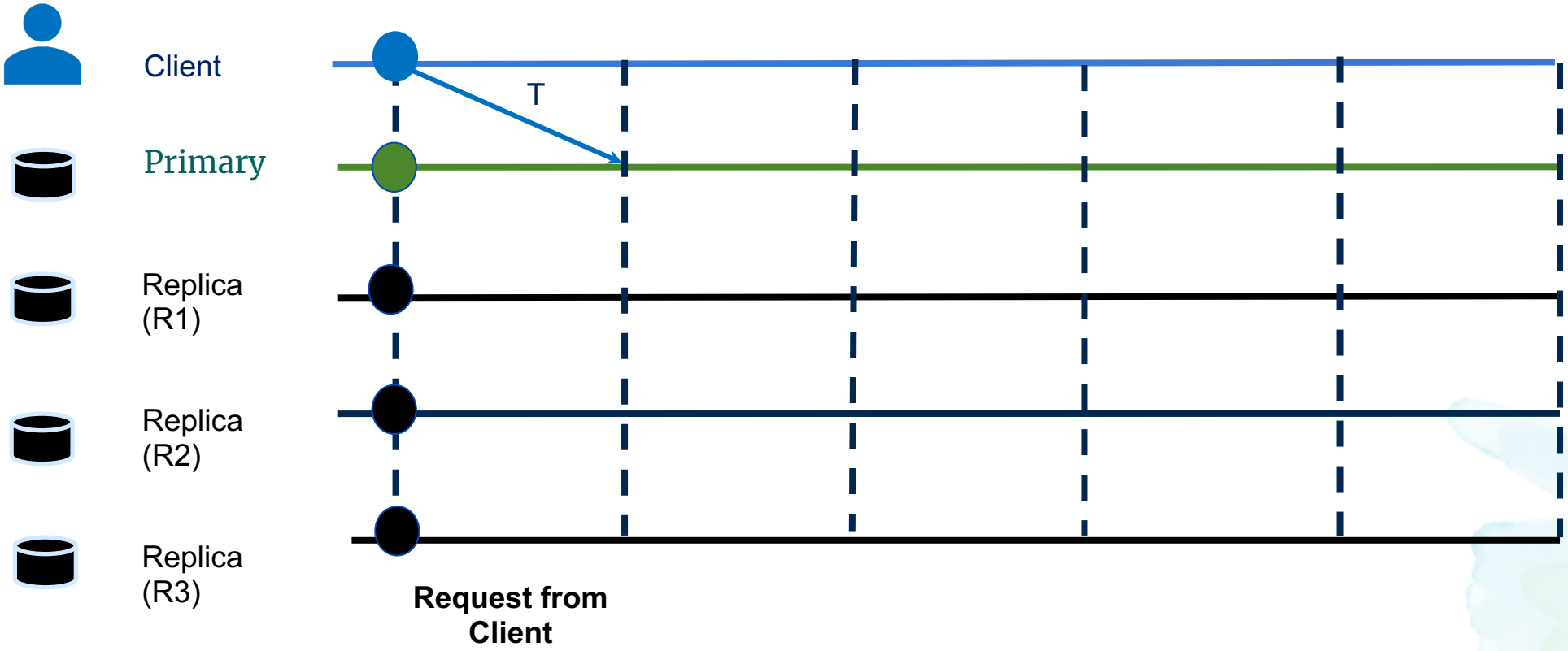


st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Understanding Throughput for PBFT

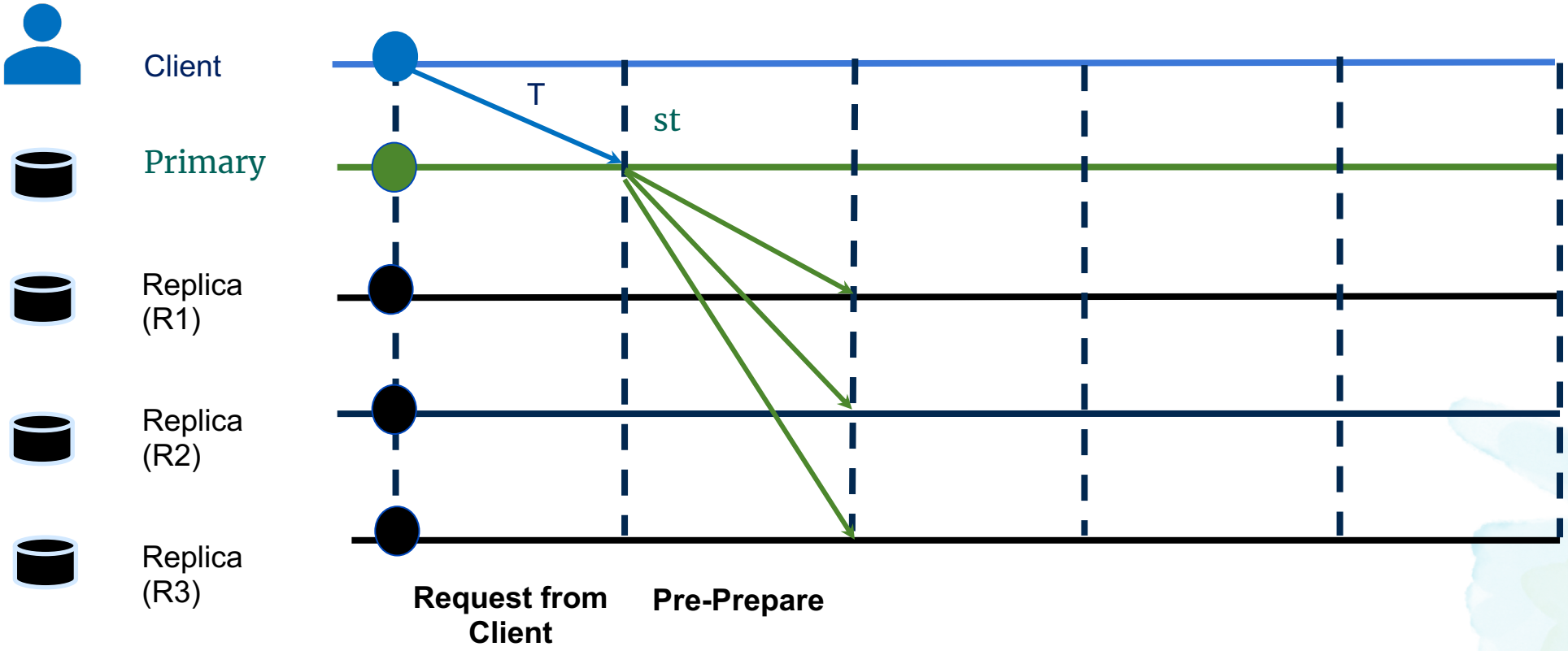


st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Understanding Throughput for PBFT

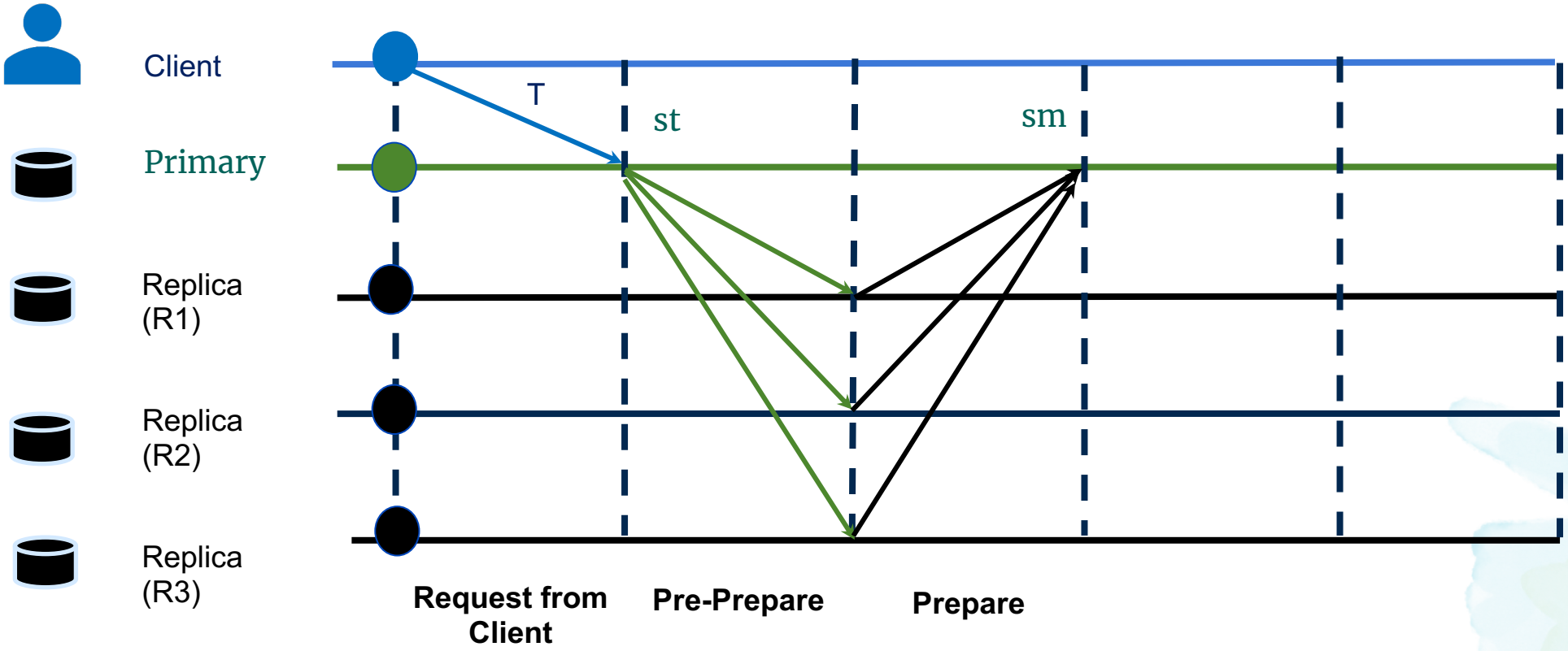


st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Understanding Throughput for PBFT

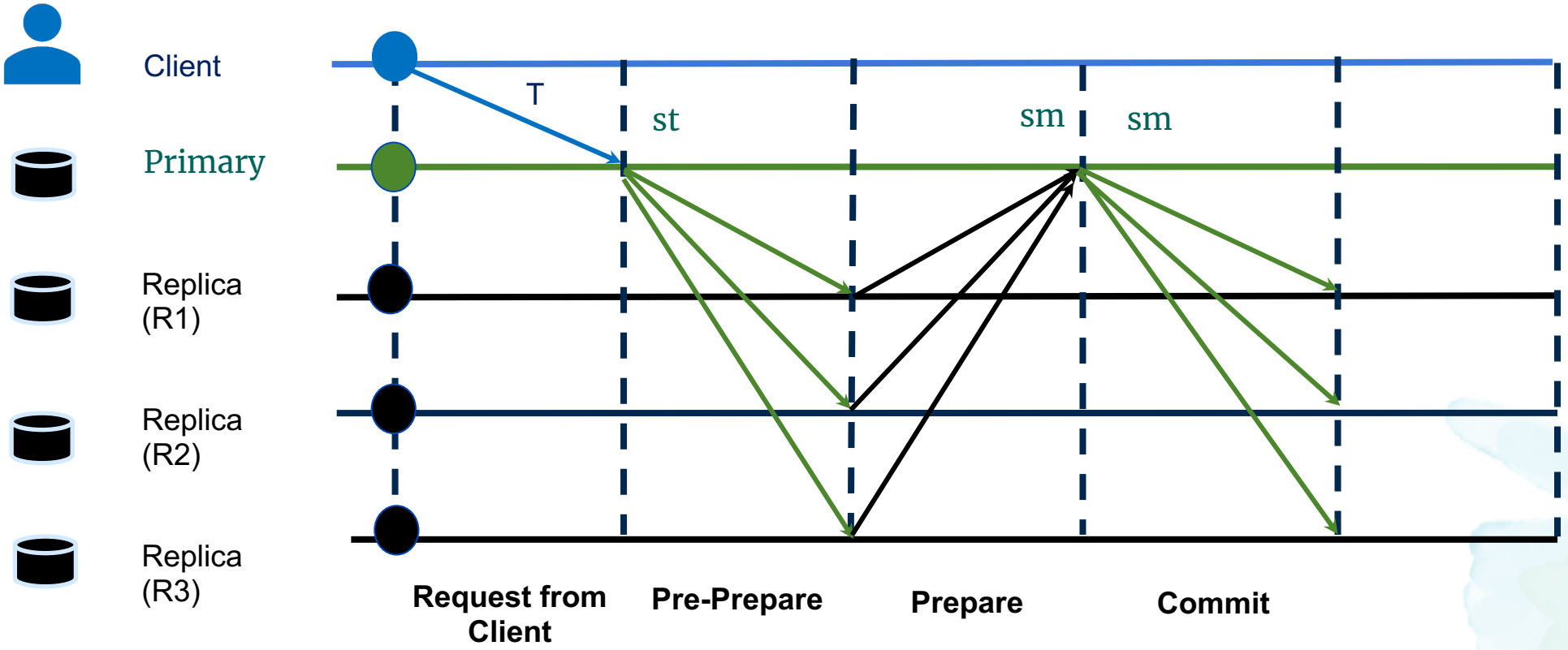


st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Understanding Throughput for PBFT

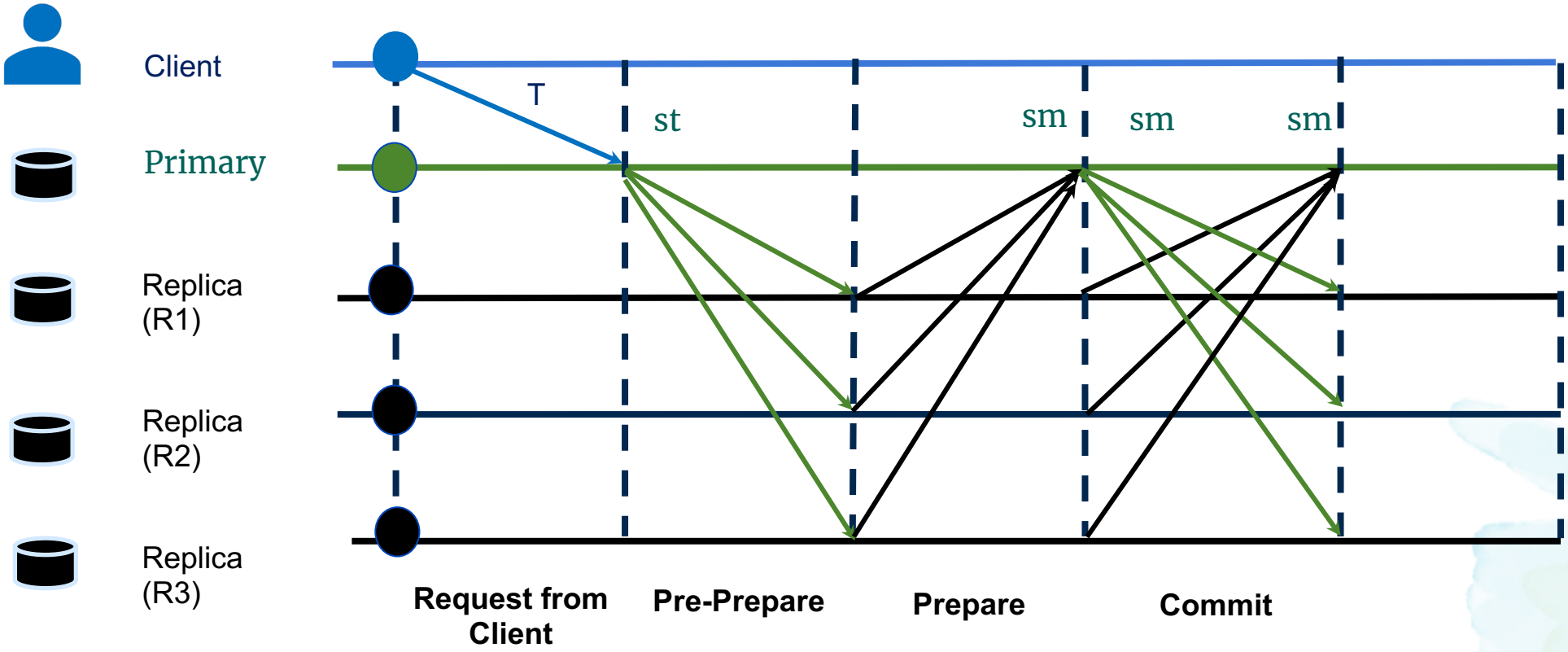


st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Understanding Throughput for PBFT

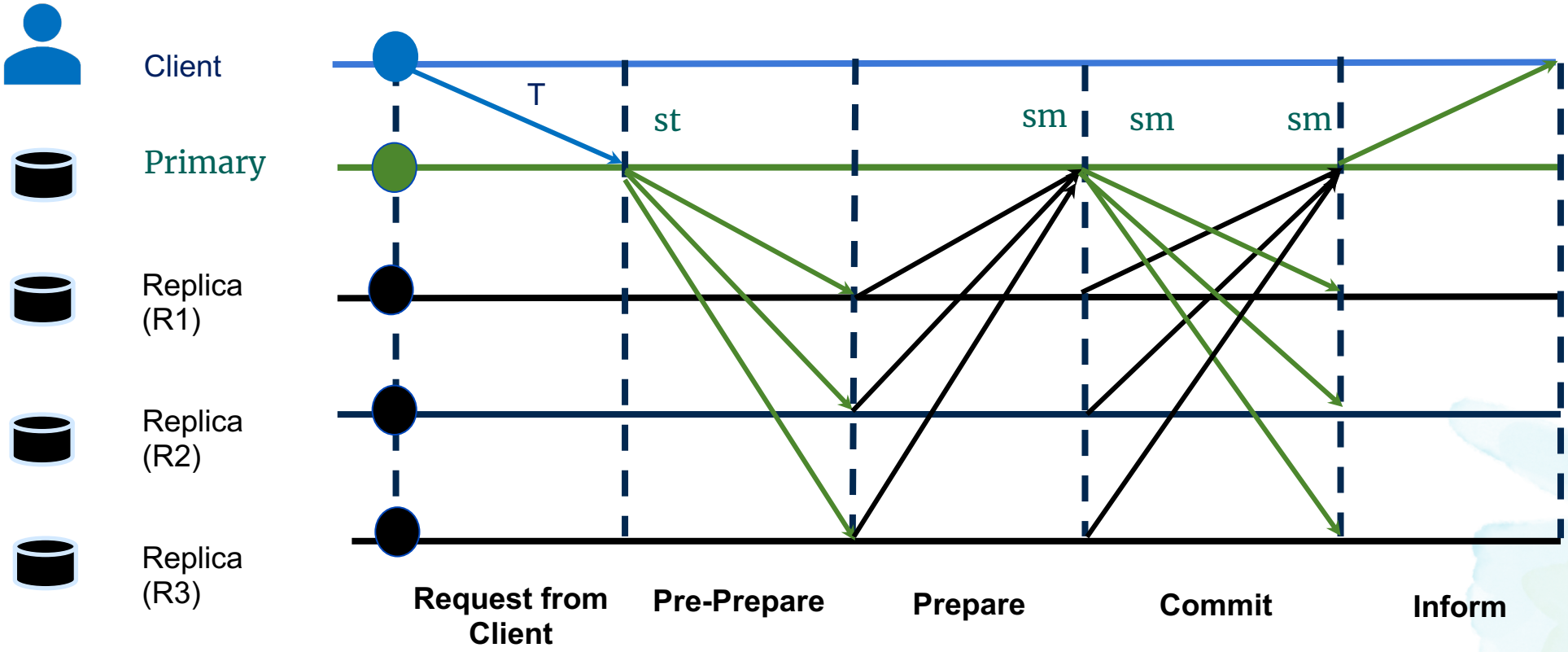


st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Understanding Throughput for PBFT



st is the the size of each transaction

sm is the size of each message being sent in PREPARE and COMMIT phase

Size of data processed by Primary in Pre-Prepare, Prepare and Commit phase
= (n-1)(st) + (n-1)sm + (n-1)sm + (n-1)sm = ((n-1)(st+3sm)) bytes

Throughput for PBFT and its Underutilization of Resourcs

Throughput for PBFT and its Underutilization of Resourcs

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

where B is the outgoing Bandwidth of the primary.

Throughput for PBFT and its Underutilization of Resourcs

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

where B is the outgoing Bandwidth of the primary.

In typical implementations of PBFT, st >> sm.

Throughput for PBFT and its Underutilization of Resourcs

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

where B is the outgoing Bandwidth of the primary.

In typical implementations of PBFT, st >> sm.

$$T_{max} = \frac{B}{(n-1)st}$$

Throughput for PBFT and its Underutilization of Resourcs

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

where B is the outgoing Bandwidth of the primary.

In typical implementations of PBFT, st >> sm.

$$T_{max} = \frac{B}{(n-1)st}$$

$$T_{PBFT} \approx T_{max}$$

Throughput for PBFT and its Underutilization of Resourcs

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

where B is the outgoing Bandwidth of the primary.

In typical implementations of PBFT, st >> sm.

$$T_{max} = \frac{B}{(n-1)st}$$

$$T_{PBFT} \approx T_{max}$$

The **primary** sends and receives roughly ***(n - 1)st*** bytes, whereas all **other replicas** only send and receive roughly ***st*** bytes.

Hence, a clear **underutilization** of other nodes.

Throughput for PBFT and its Underutilization of Resourcs

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

where B is the outgoing Bandwidth of the primary.

In typical implementations of PBFT, st >> sm.

$$T_{max} = \frac{B}{(n-1)st}$$

$$T_{PBFT} \approx T_{max}$$

The **primary** sends and receives roughly ***(n - 1)st*** bytes, whereas all **other replicas** only send and receive roughly ***st*** bytes.

Hence, a clear **underutilization** of other nodes.

Limitation

Proposed solution

Proposed solution

Key insight: Better utilization of available resources, using many primaries that *concurrently* propose transactions.

Proposed solution

Key insight: Better utilization of available resources, using many primaries that *concurrently* propose transactions.

- Proposal of **Concurrent consensus** and show that concurrent consensus can achieve much **higher throughput** than primary-backup consensus by effectively utilizing all available system resources.
- **RCC**, a paradigm for turning any primary-backup consensus protocol into a concurrent consensus protocol and that is designed for maximizing throughput.
- RCC can be utilized to make systems more **resilient**, as it can mitigate the effects of order-based attacks and throttling attacks

Proposed solution

Key insight: Better utilization of available resources, using many primaries that *concurrently* propose transactions.

- Proposal of **Concurrent consensus** and show that concurrent consensus can achieve much **higher throughput** than primary-backup consensus by effectively utilizing all available system resources.
- **RCC**, a paradigm for turning any primary-backup consensus protocol into a concurrent consensus protocol and that is designed for maximizing throughput.
- RCC can be utilized to make systems more **resilient**, as it can mitigate the effects of order-based attacks and throttling attacks

The Promise of Concurrent Consensus

Key ideas

- **Democracy -** Give all the replicas the power to be the primary.
- **Parallelism -** Run multiple parallel instances of a BFT protocol. Each instance has a primary.
- **Decentralization -** Always there will be a set of ordered client requests.

Throughput for Concurrent PBFT

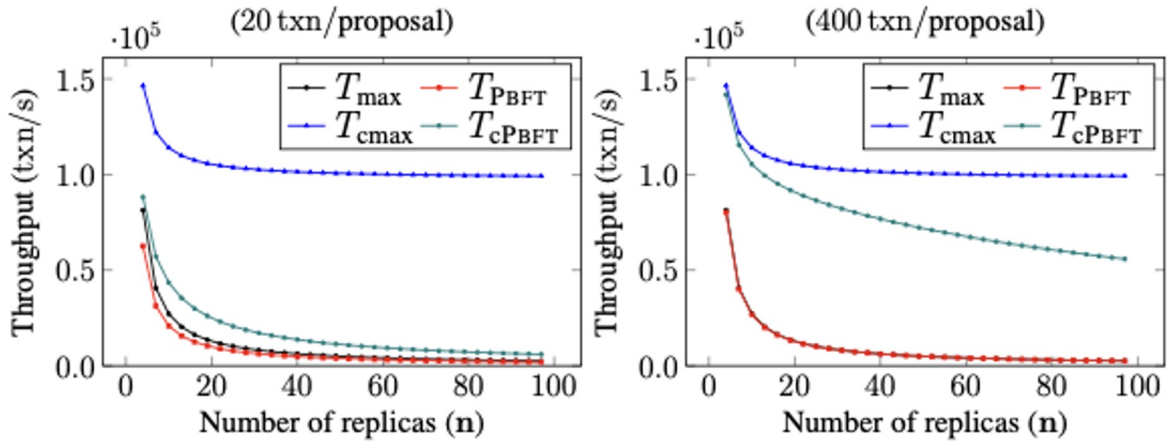


Fig. 1. Maximum throughput of replication in a system with $B = 1\,\text{Gbit/s}$, $\mathbf{n} = 3\mathbf{f} + 1$, $\mathbf{nf} = 2\mathbf{f} + 1$, $sm = 1\,\text{KiB}$, and individual transactions are 512 B. On the *left*, each proposal groups 20 transactions ($st = 10\,\text{KiB}$) and on the *right*, each proposal groups 400 transactions ($st = 2\,\text{MiB}$).

nf→non-faulty primaries
st→Size of a transaction
sm→ size of prepare/commit messages

With no concurrency

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

$$T_{max} = \frac{B}{(n-1)st}$$

$$T_{PBFT} \approx T_{max}$$

With concurrency

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

$$T_{cPBFT} = \frac{\text{nf. B}}{(((\text{n} - 1)\text{st} + 3\text{sm}) + (\text{nf} - 1)(\text{st} + 4(\text{n} - 1)\text{sm})}$$

Design Goals

Design Goals

- Provide Consensus among replicas on the client transactions and the order in which they are to be executed.
- RCC is a **Design Paradigm** that can be applied to any other primary-backup BCA making it concurrent.
- Dealing with faulty primaries does not interfere with the operations of other consensus-instances.

Before proceeding: Assumptions

Before proceeding: Assumptions

A1) If no failures are detected in round **ρ** of BCA (the round is successful), then at least **nf-f** non-faulty replicas have accepted a proposed transaction in round **ρ**.

A2) If a non-faulty replica accepts a proposed transaction **T** in round **ρ** of BCA, then all other non-faulty replicas that accepted a proposed transaction, accepted **T**.

A3) If a non-faulty replica accepts a transaction **T** , then **T** can be recovered from the state of any subset of **nf-f** non-faulty replicas.

A4) If the primary is non-faulty and communication is reliable, then all non-faulty replicas will accept a proposal in round **ρ** of BCA.

Design of RCC

Design of RCC

Concurrent BCA

- Each replica participates in **m** instances of the BCA; $(1 \leq m \leq n)$
- In each instance, a transaction is proposed by a replica acting as Primary for that instance.

Design of RCC

Concurrent BCA

- Each replica participates in **m** instances of the BCA; ($1 \leq m \leq n$)
- In each instance, a transaction is proposed by a replica acting as Primary for that instance.

Ordering

- Since instances are running in parallel, ordering before execution is crucial.
- A deterministic global order is created for all the transactions.

Design of RCC

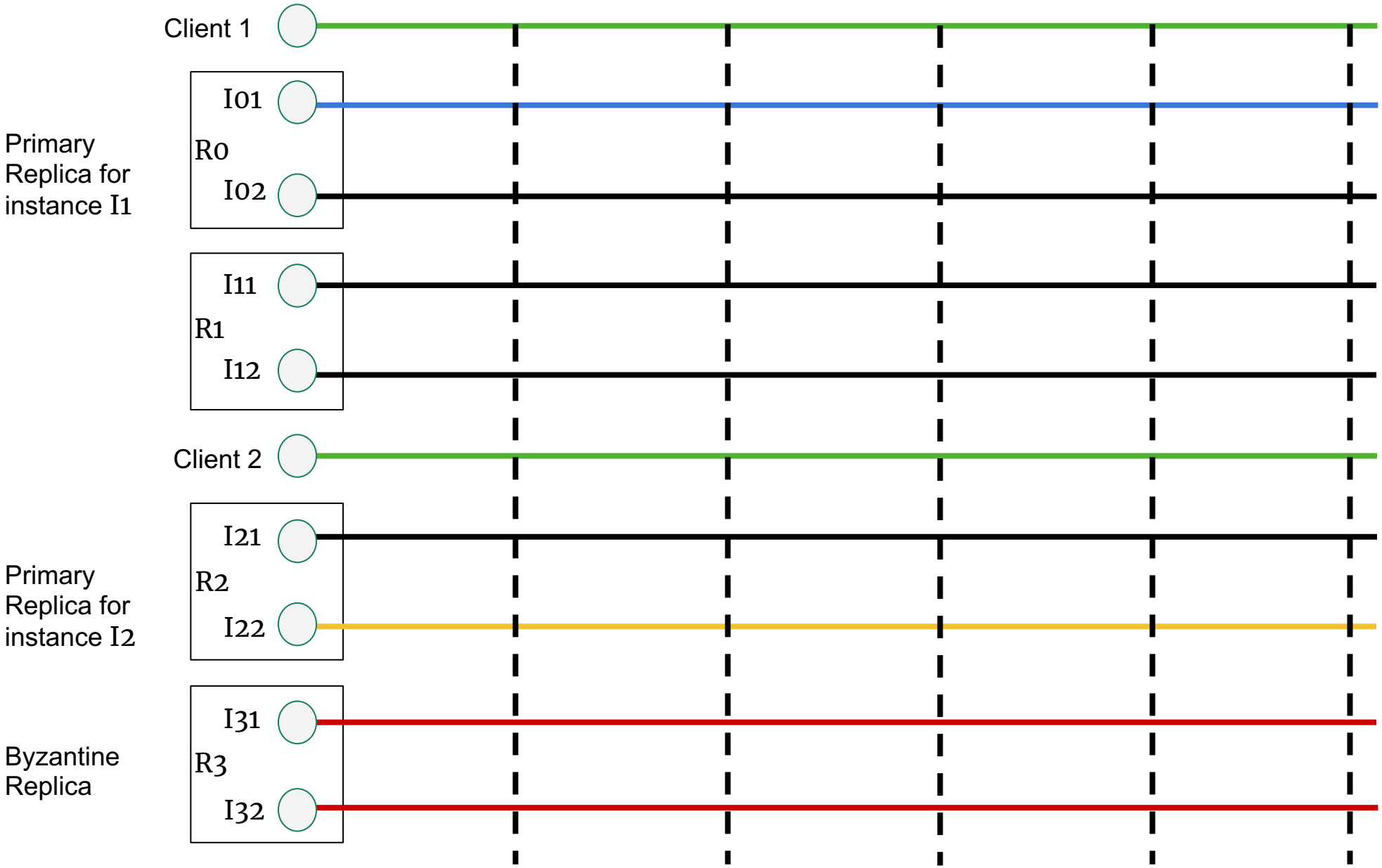
Concurrent BCA → Ordering → Execution

- Each replica participates in **m** instances of the BCA; ($1 \leq m \leq n$)
- In each instance, a transaction is proposed by a replica acting as Primary for that instance.

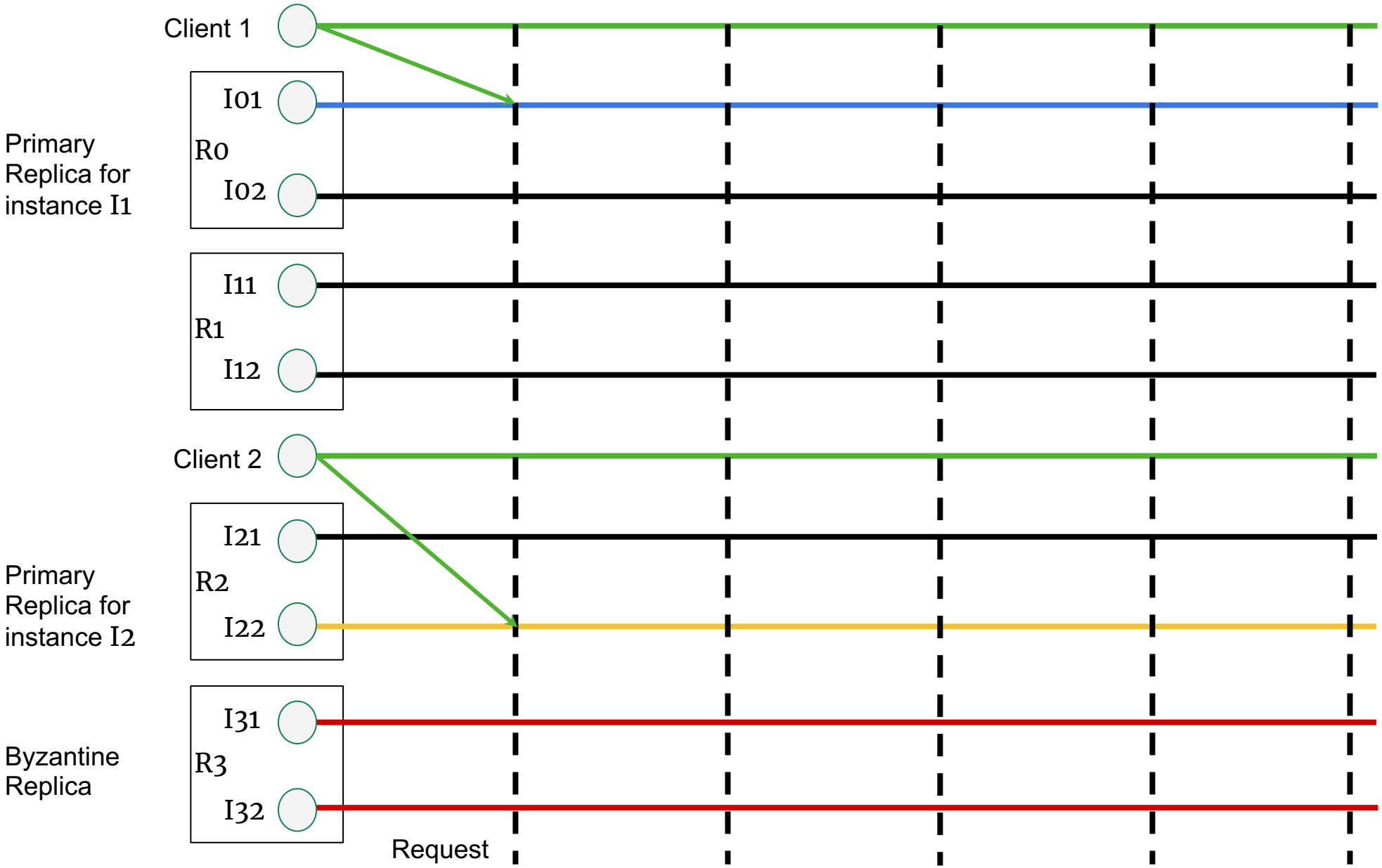
- Since instances are running in parallel, ordering before execution is crucial.
- A deterministic global order is created for all the transactions.

- Each replica executes the transactions in that round in the global order and informs the client.

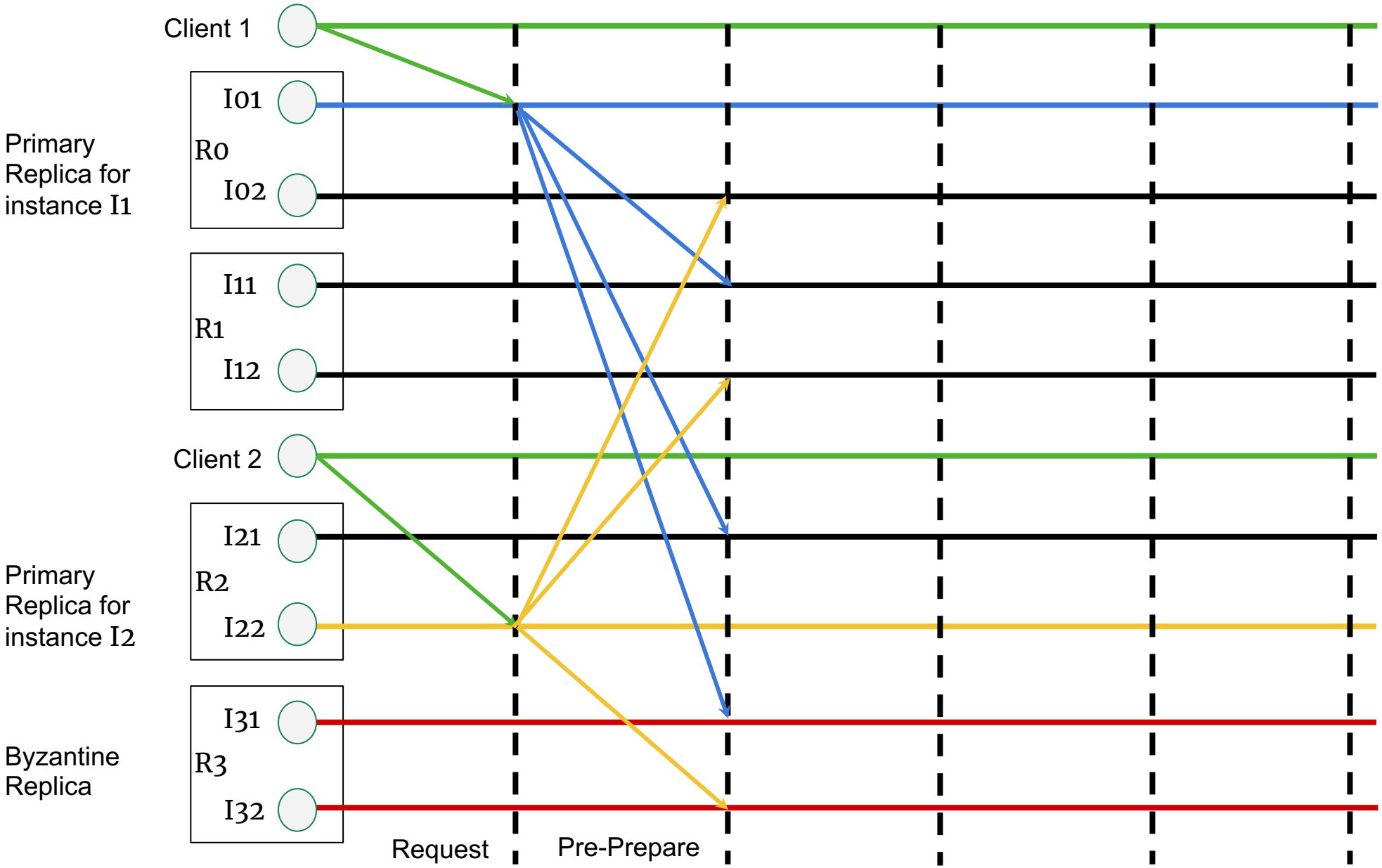
Example: RCC over PBFT



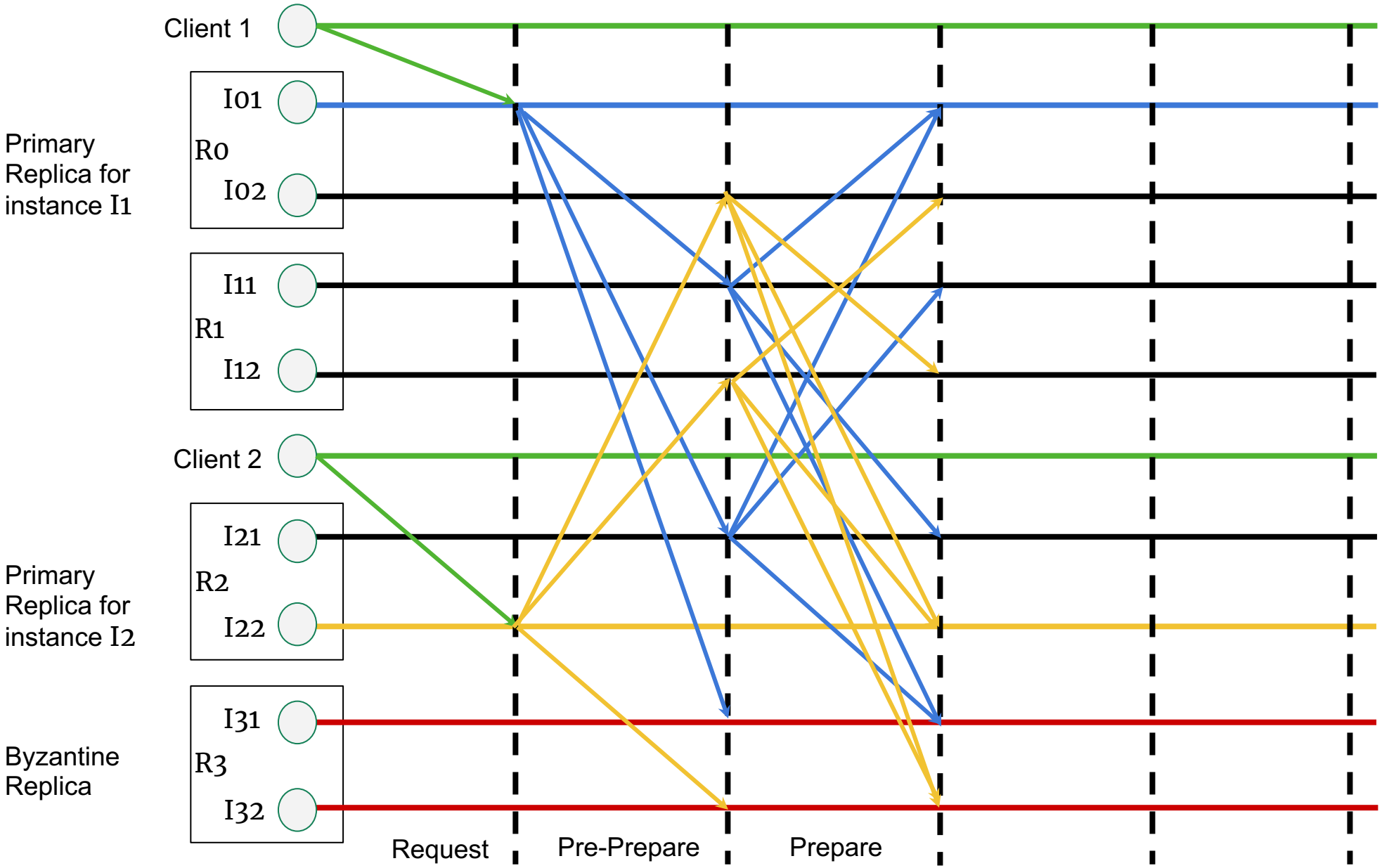
Example: RCC over PBFT



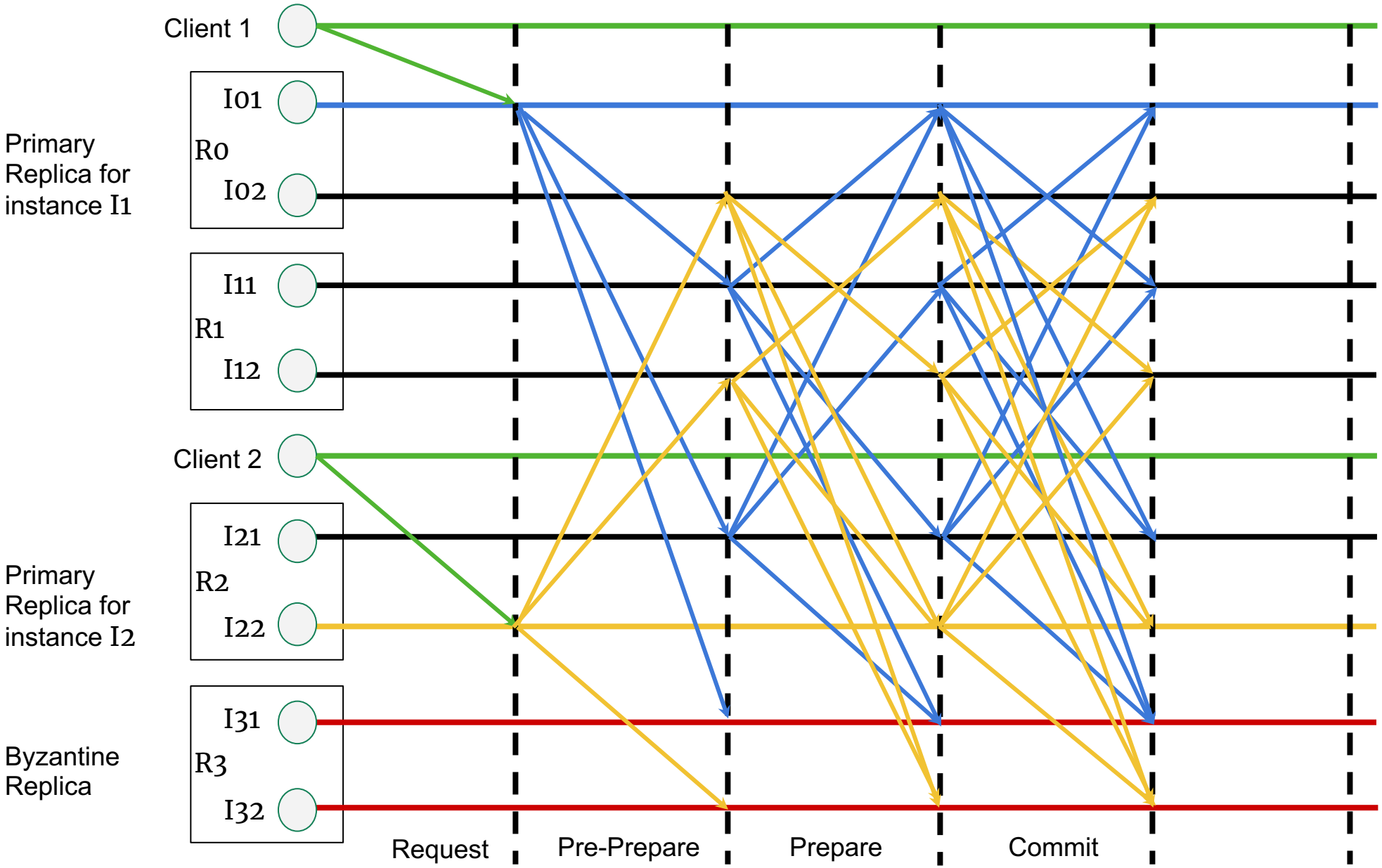
Example: RCC over PBFT



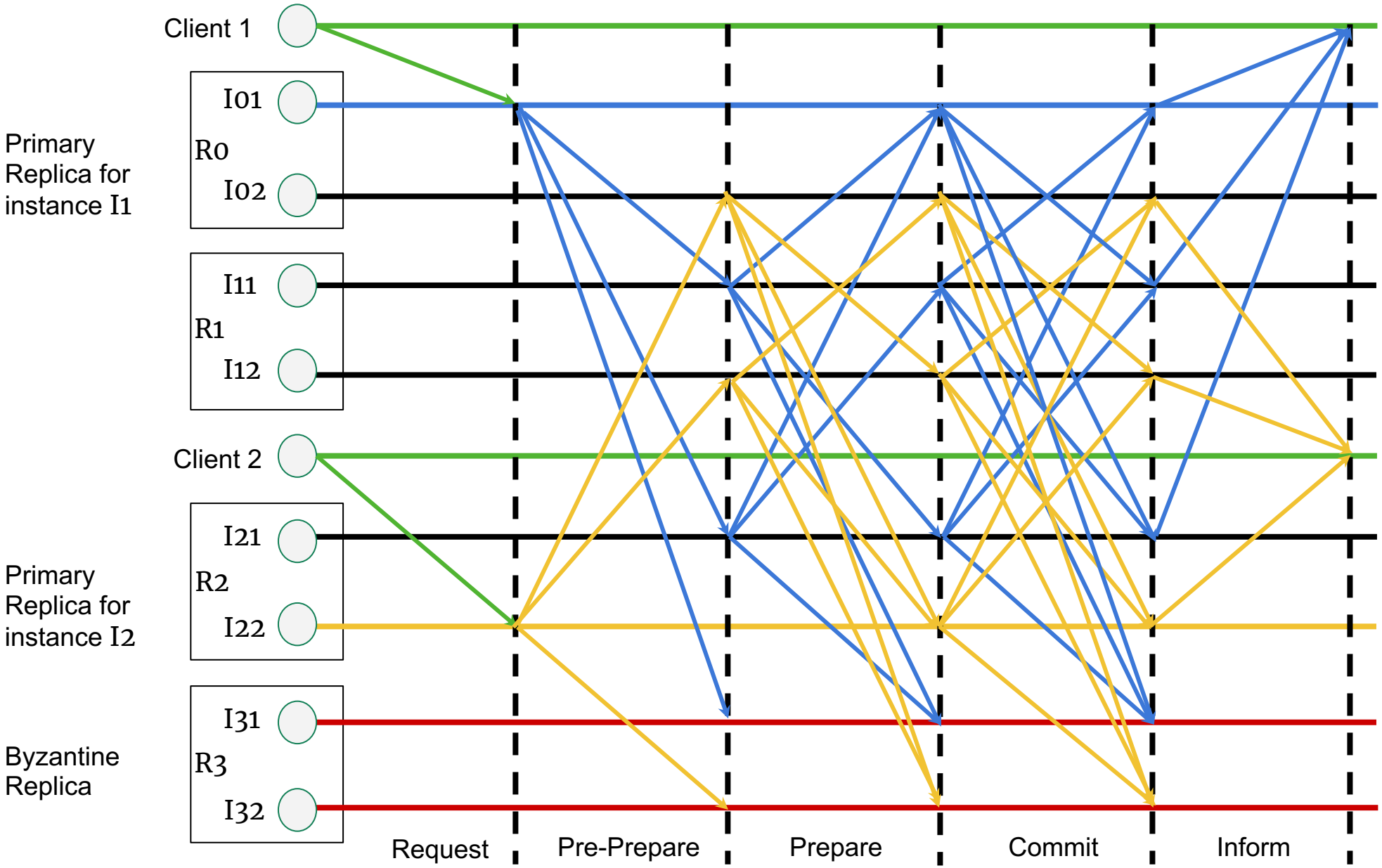
Example: RCC over PBFT



Example: RCC over PBFT



Example: RCC over PBFT



Failures in RCC

Consensus based systems operate in ***asynchronous environments***, i.e., messages can get lost or arrive with arbitrary delays or orders.

It is impossible to distinguish between a malicious primary that doesn't send out proposals and a primary that does send out proposals but gets lost in the network.

These asynchronous consensus protocols progress only in a period of ***reliable bounded-delay communication***, i.e., with some maximum delay, all messages sent by non-faulty replicas (nf) will reach their destination.

Types of Failures in RCC

- **Detectable Failures**

- **Undetectable Failures**

Dealing with Detectable Failures

To deal with detectable failures, RCC assumes that failure of non-faulty replicas to receive proposals from a primary is due to the fault of primary.

RCC detects the failure and recovers from it without interfering with other BCA instances or the recovery process, hence adhering to the wait free design.

Recovery Protocol

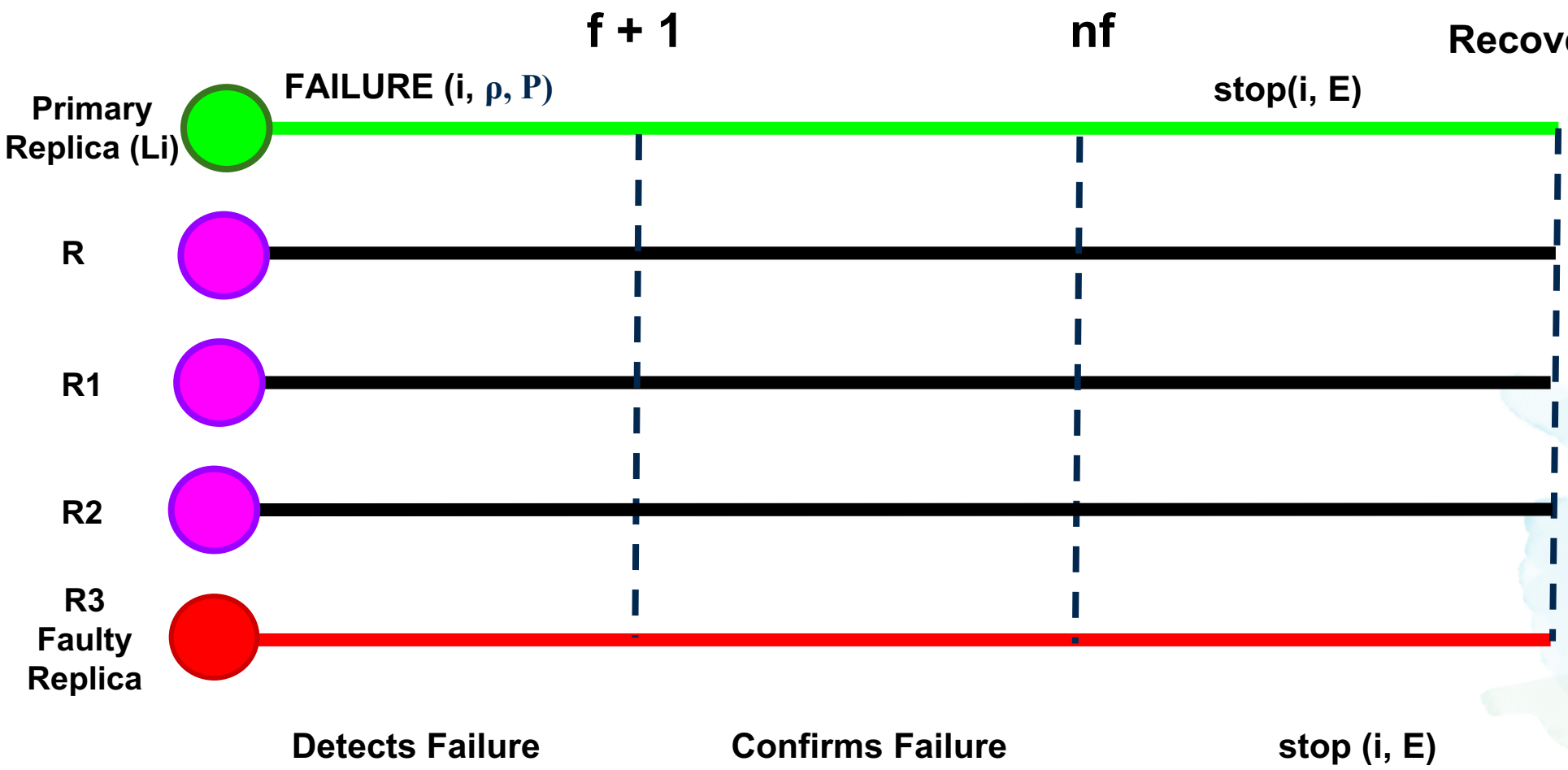
Note: Assume that primary (**P_i**) of Instance I_i fails ***[1<= i <= m]***

The recovery protocol follows 3 steps:

- All replicas will detect the failure of **P_i**
- All replicas will reach agreement on state of I_i
- All replicas will determine which state to resume its operations

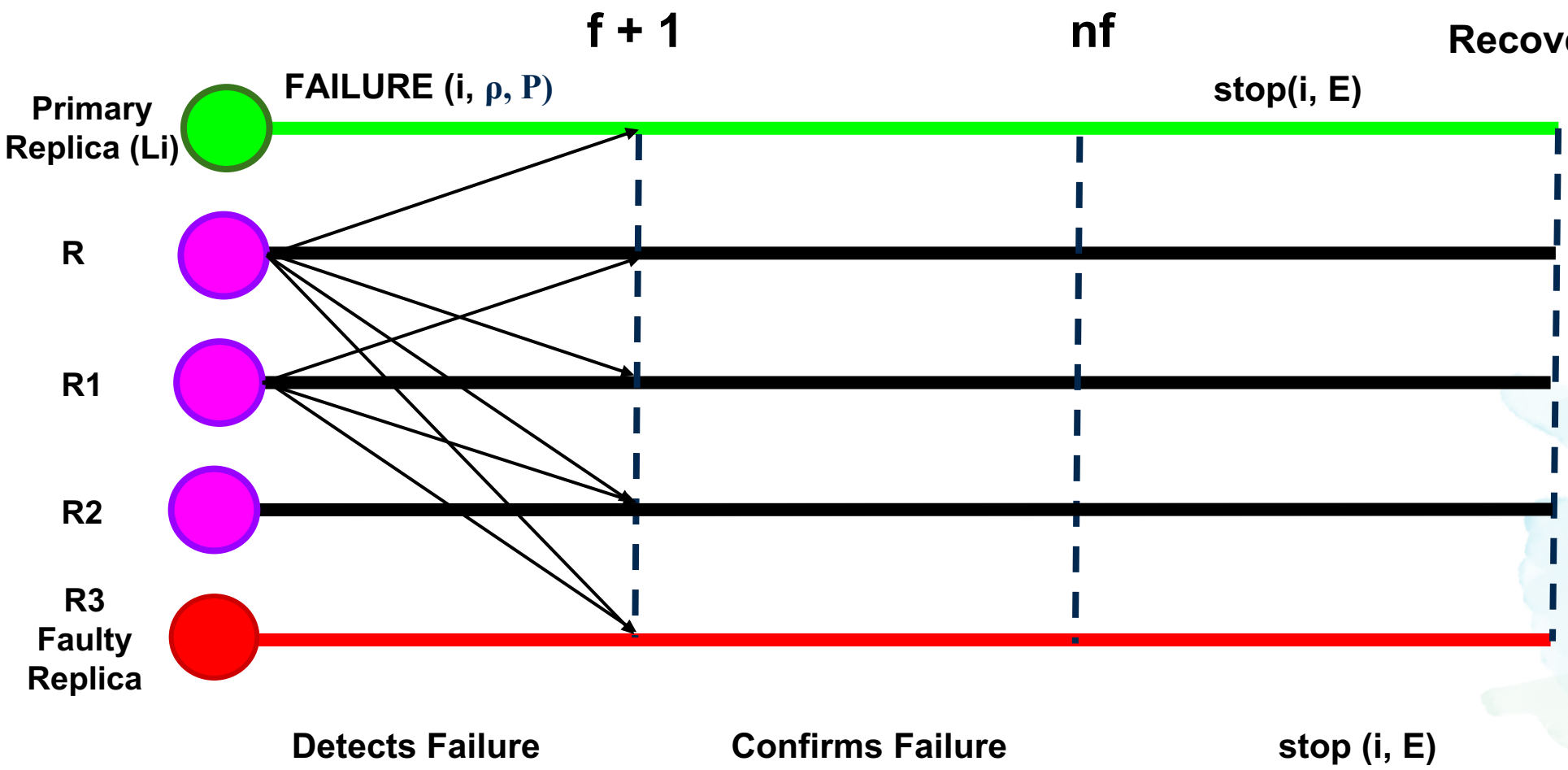
A separate primary backup protocol, whose primary is **L_i**, will be used to reach an agreement on the state of I_i

Dealing with Detectable Failures



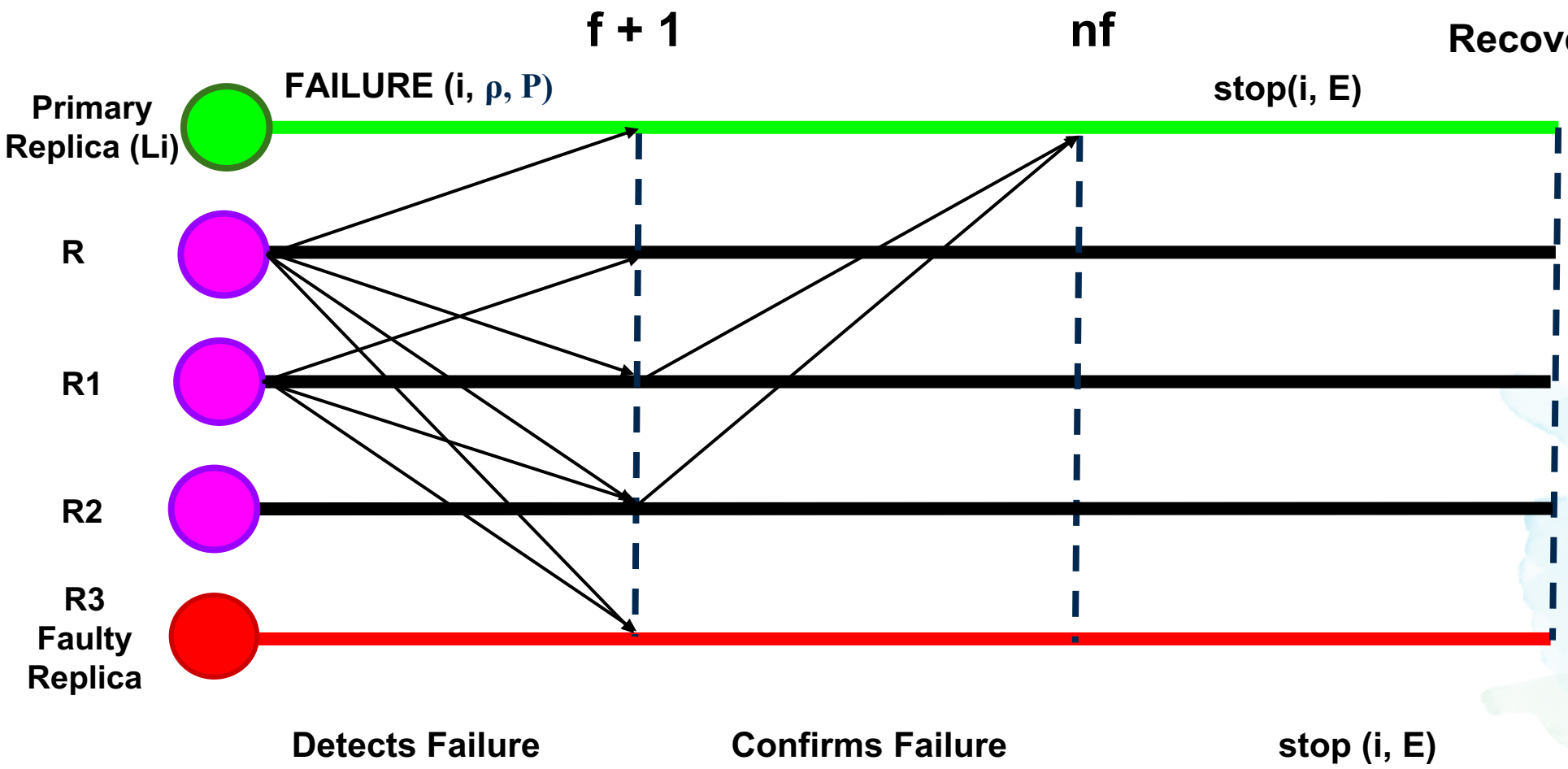
I_i: i-th Instance
ρ: Round
P: state of R
E: set of nf failure messages

Dealing with Detectable Failures



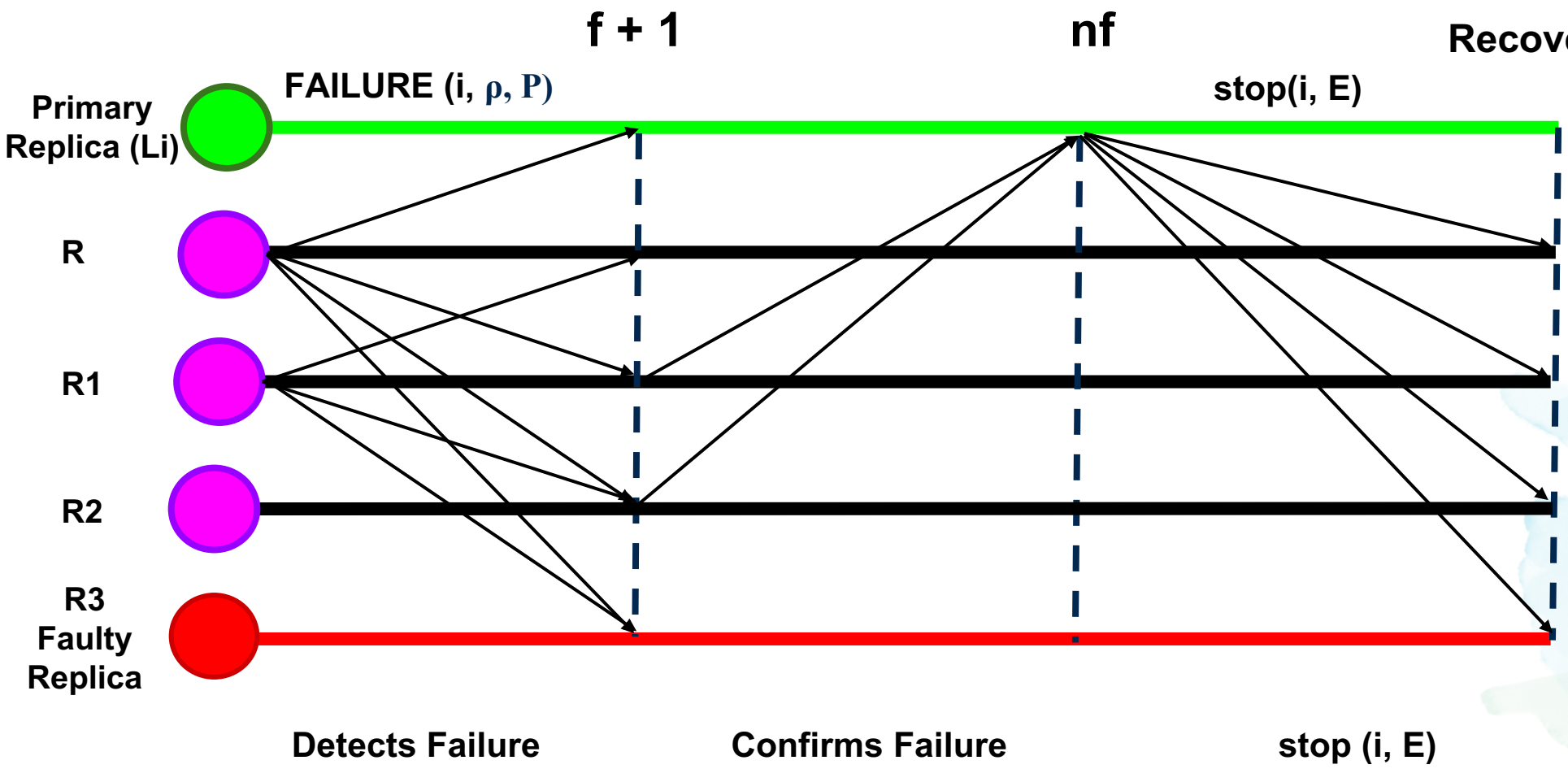
I_i: i-th Instance
ρ: Round
P: state of R
E: set of nf failure messages

Dealing with Detectable Failures



I_i: i-th Instance
ρ: Round
P: state of R
E: set of nf failure messages

Dealing with Detectable Failures



I_i: i-th Instance
ρ: Round
P: state of R
E: set of nf failure messages

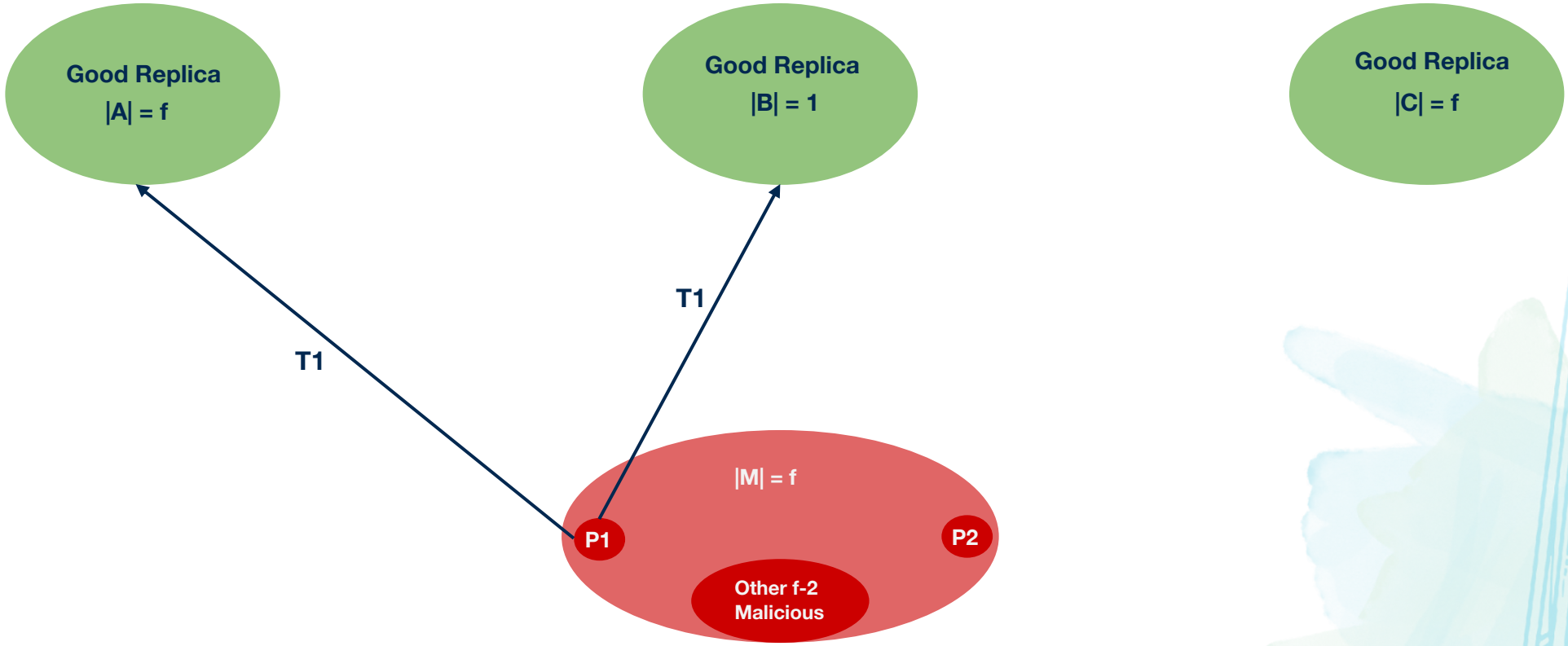
Dealing with Undetectable Failures

- Consider a system with **n = 3f+1 =** 7 replicas.
- Assume **P1** and **P2** are malicious.
- We partition the non faulty replicas into 3 sets: **A , B , C**

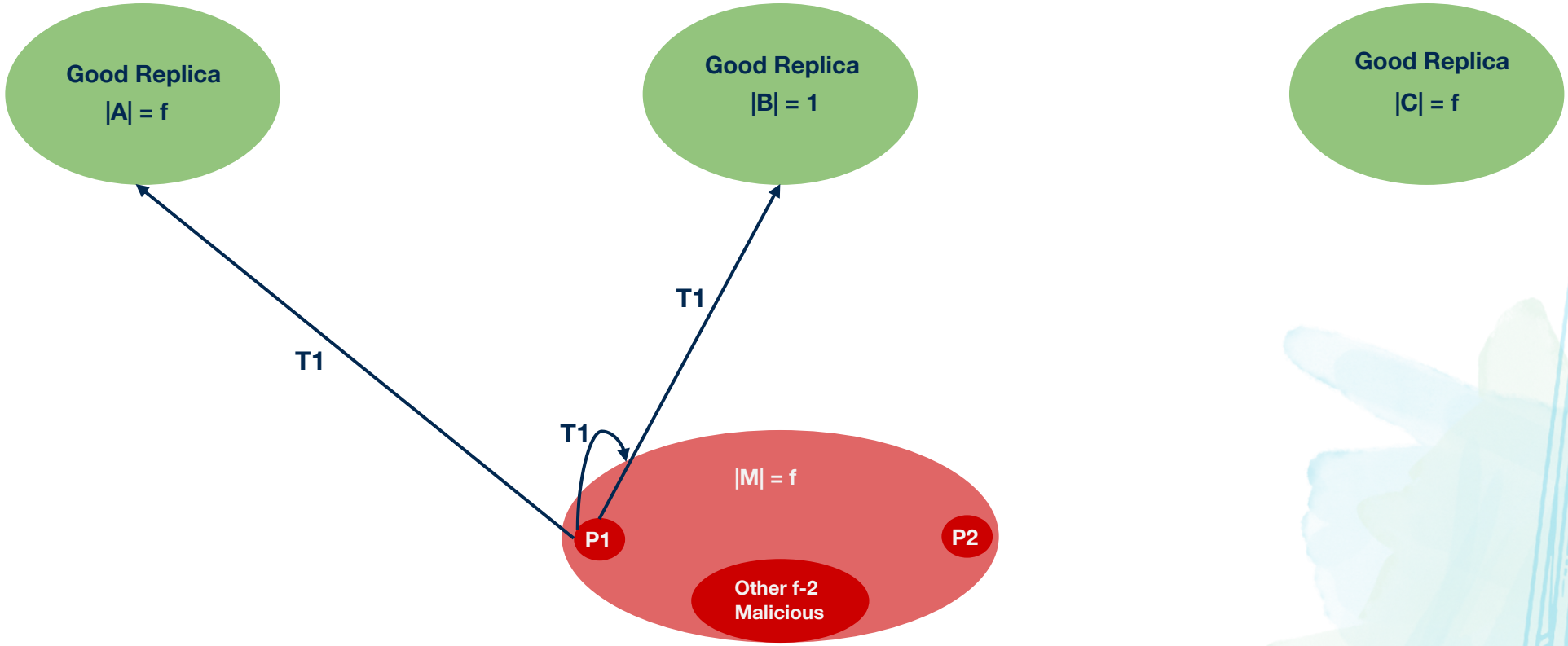
Dealing with Undetectable Failures

Dealing with Undetectable Failures

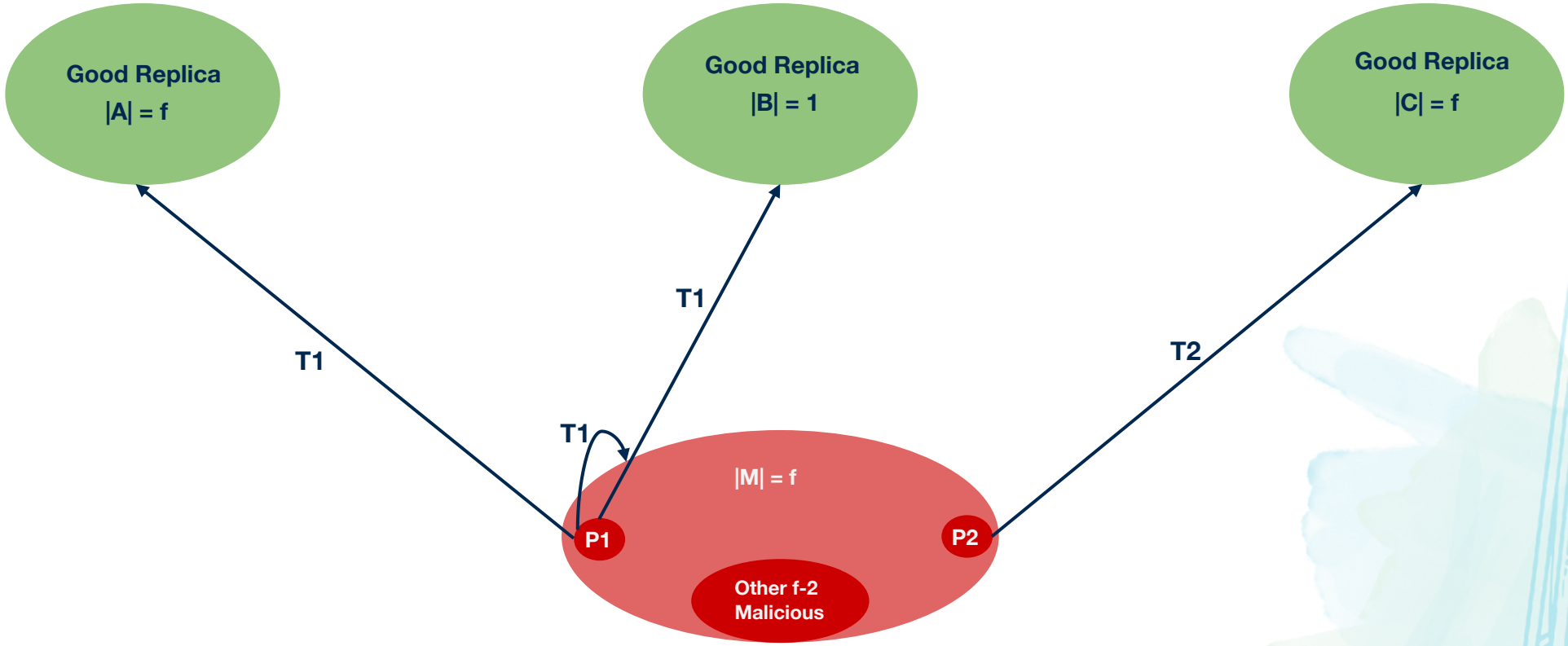
Dealing with Undetectable Failures



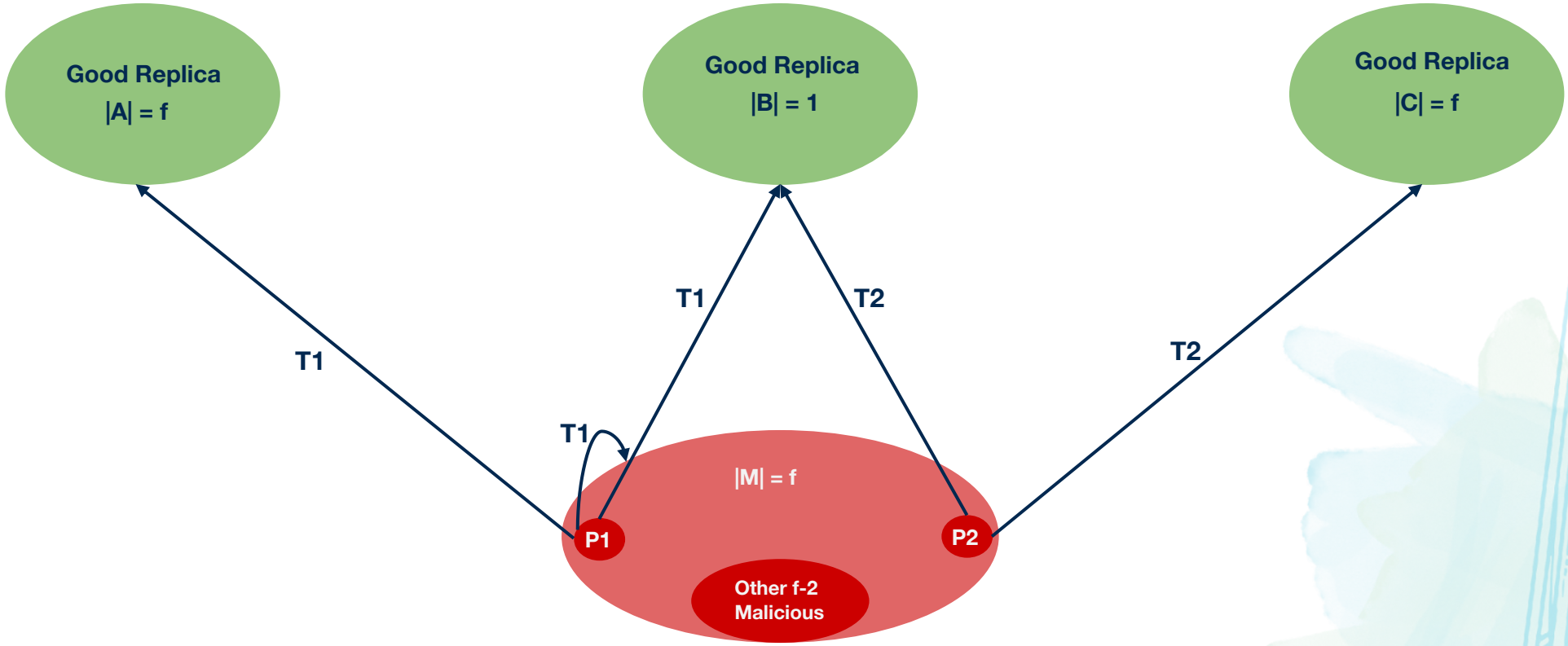
Dealing with Undetectable Failures



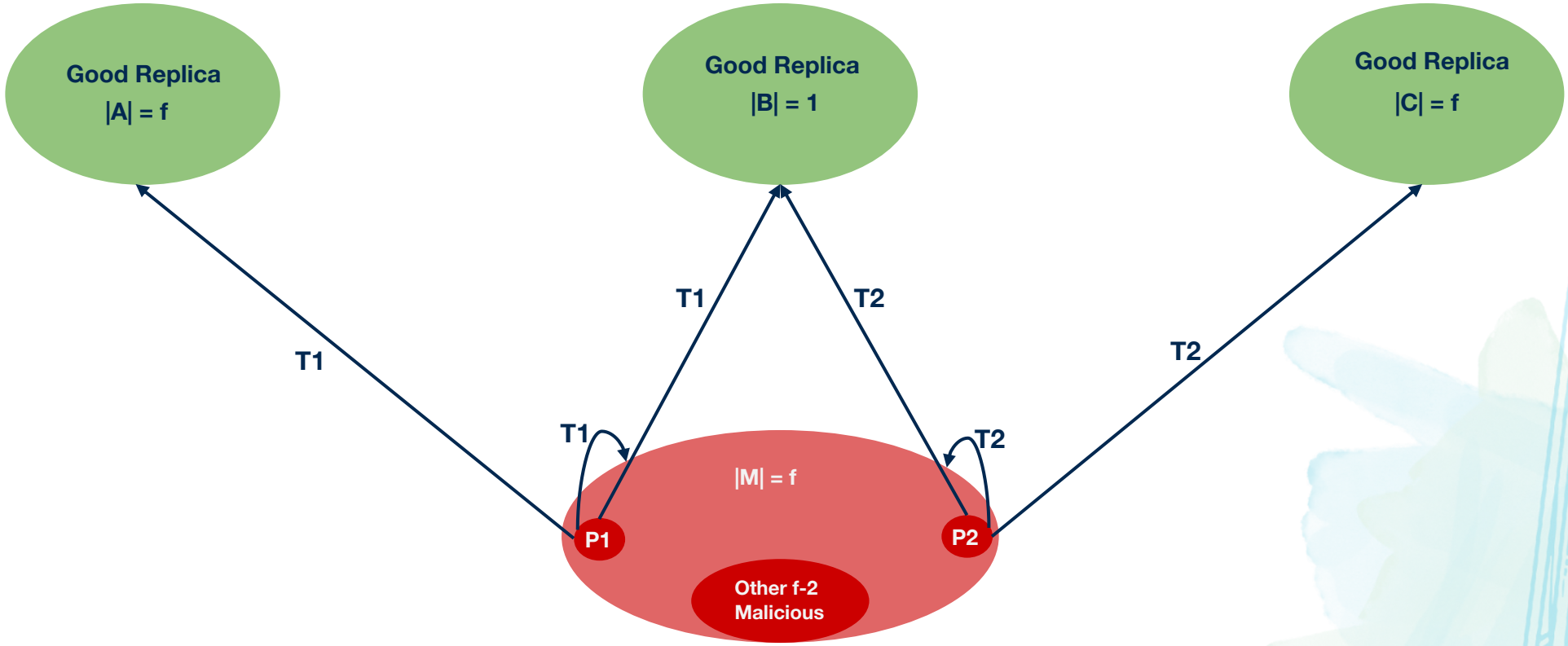
Dealing with Undetectable Failures



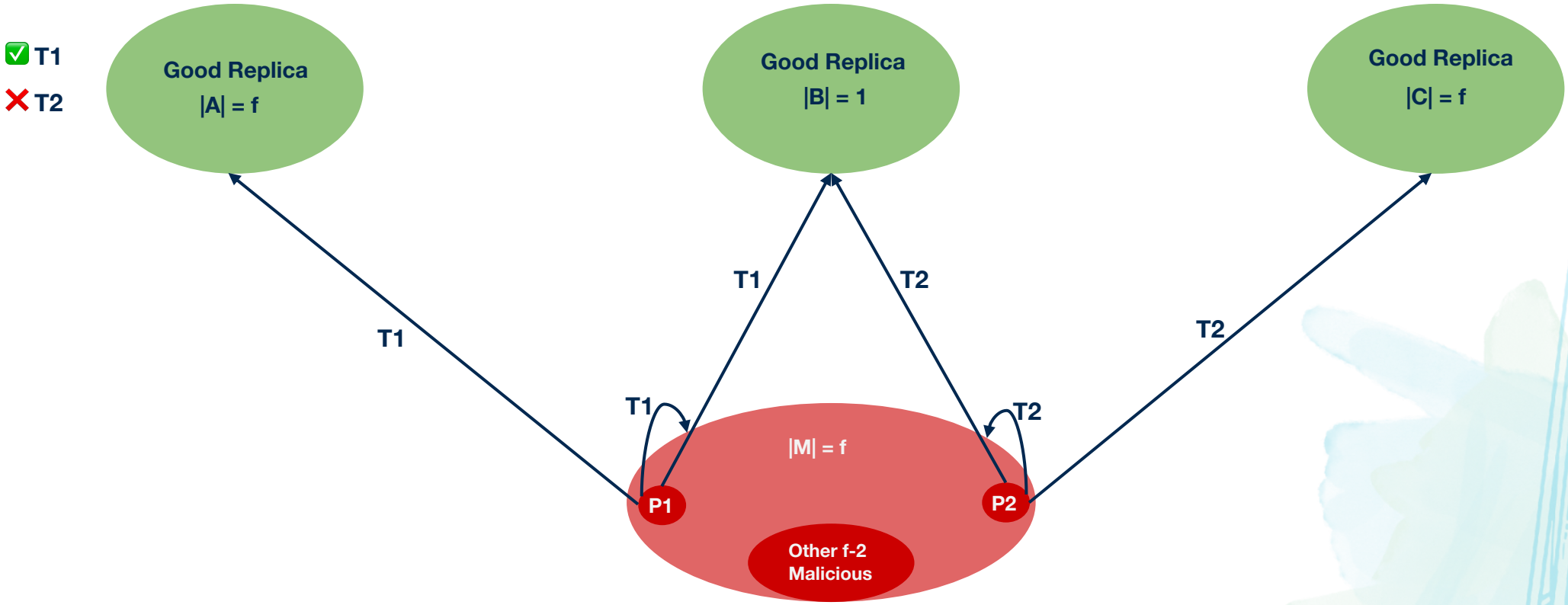
Dealing with Undetectable Failures



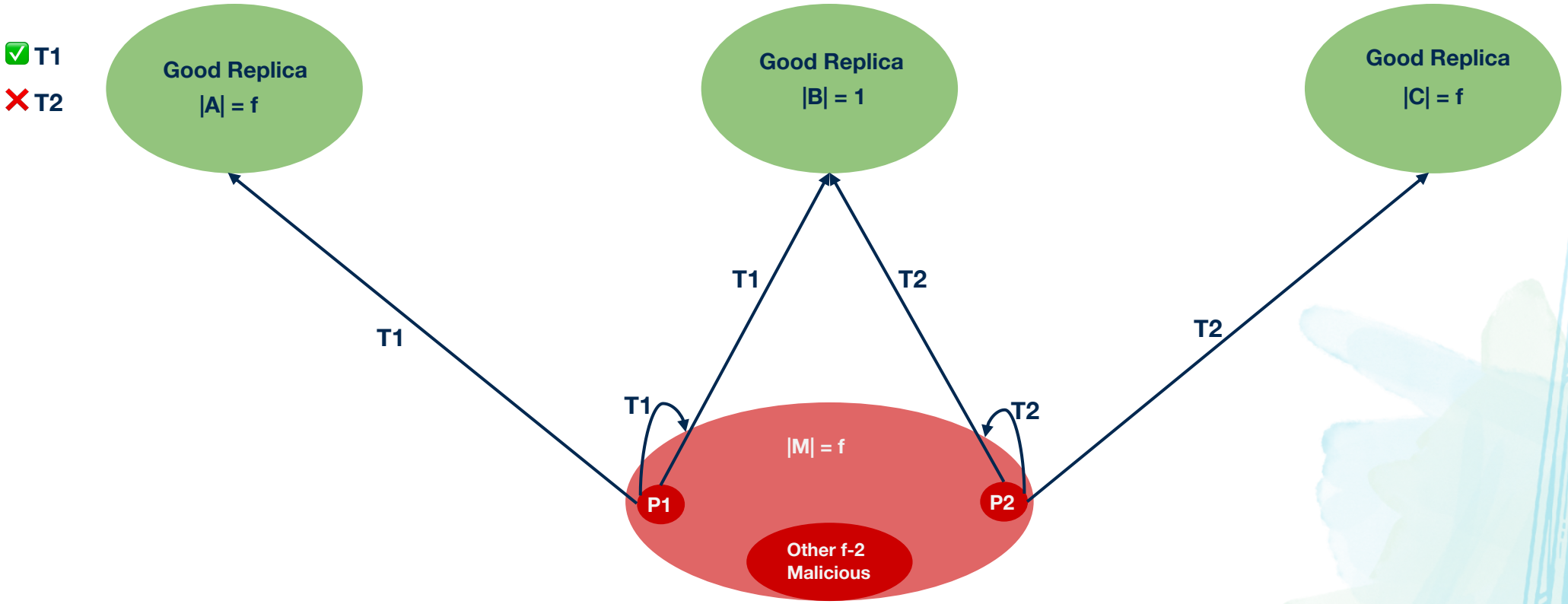
Dealing with Undetectable Failures



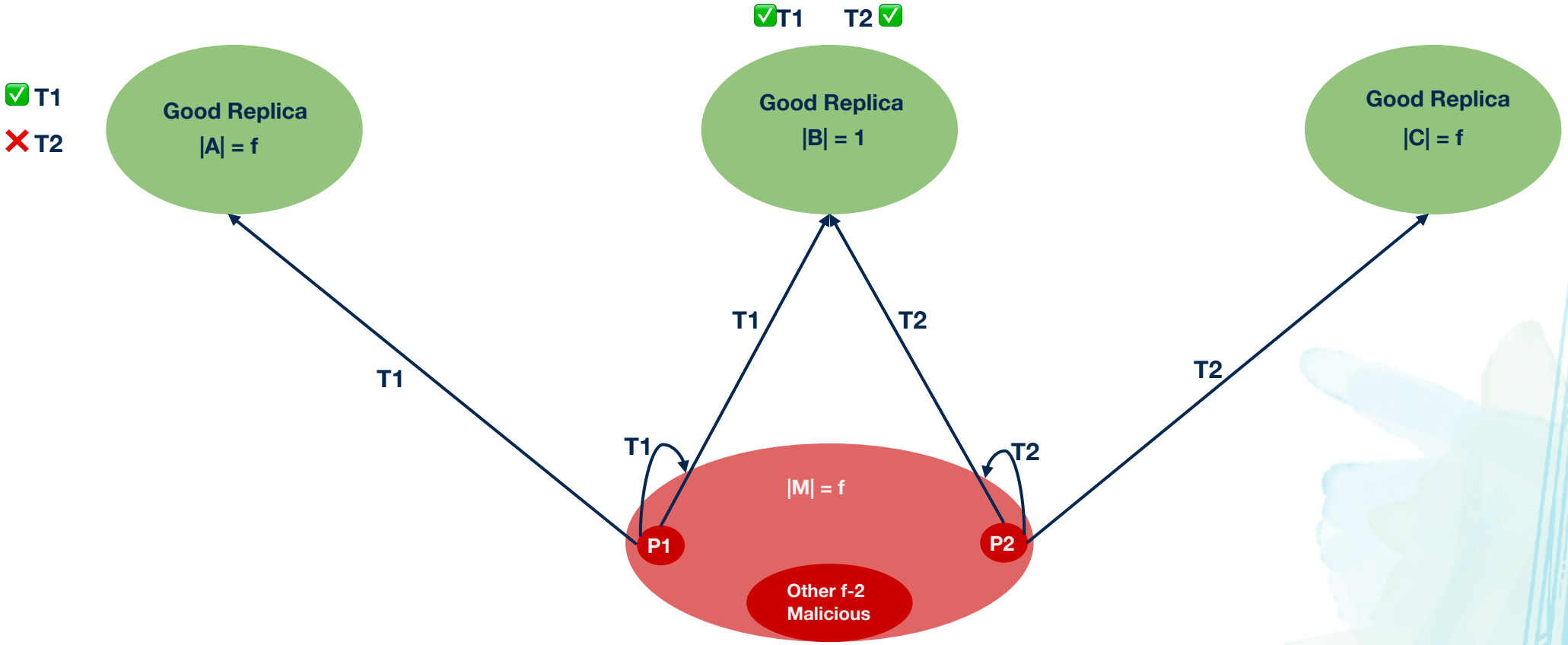
Dealing with Undetectable Failures



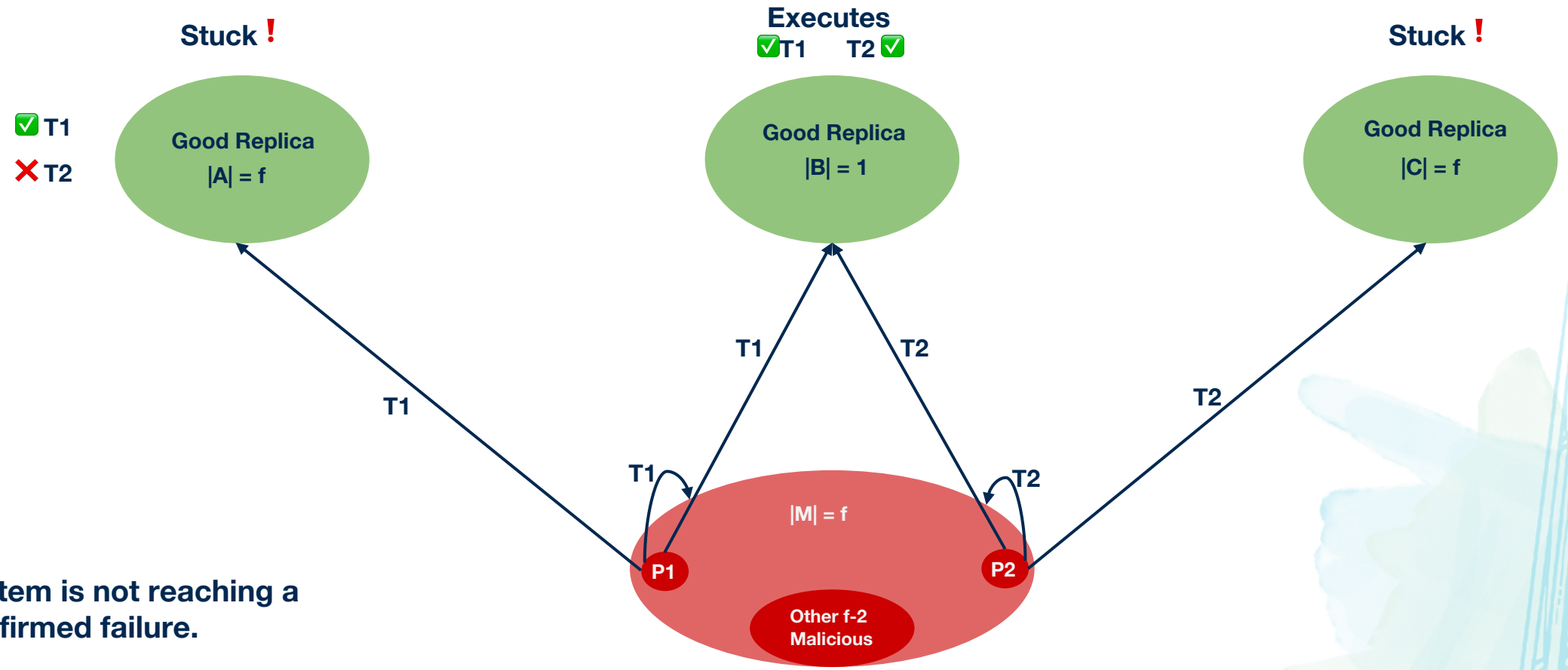
Dealing with Undetectable Failures



Dealing with Undetectable Failures



Dealing with Undetectable Failures



System is not reaching a confirmed failure.

Deal with these in-the-dark attacks

Solution ?

Deal with these in-the-dark attacks

Solution ?

A checkpoint algorithm! ✅

Revisiting PBFT Checkpoint Algorithm

- To discard messages from the replica's log. (But keep these messages in log until f+1 non-faulty replicas have executed the associated transaction)
- Algorithm employs periodic checkpoints – after every 100^{th} sequence numbered transaction(n).

- This checkpoint contains a proof of stability:How?

1. **Checkpoint Creation**: When a checkpoint is created, it is multicast to other replicas along with its digest.
2. **Proof Collection**: Each replica collects checkpoint messages until it has a sufficient number (2f+1) to prove the correctness of the checkpoint.

We will refer to the states produced by the execution of these requests as checkpoints and we will say that a **checkpoint with a proof is a stable checkpoint.**

Once a checkpoint has been verified and deemed stable, the replica can discard all **pre-prepare, prepare, and commit** messages with sequence numbers less than or equal to that of the checkpoint.

Revisiting Assumptions A1 , A2 , A3

Revisiting Assumptions A1 , A2 , A3

A1: If no failures are detected in round **ρ** of BCA (the round is successful), then at least **nf-f** non-faulty replicas have accepted a proposed transaction in round **ρ**.

Revisiting Assumptions A1 , A2 , A3

A1: If no failures are detected in round **ρ** of BCA (the round is successful), then at least **nf-f** non-faulty replicas have accepted a proposed transaction in round **ρ**.

A2: If a non-faulty replica accepts a proposed transaction **T** in round **ρ** of BCA, then all other non-faulty replicas that accepted a proposed transaction, accepted **T**.

Revisiting Assumptions A1 , A2 , A3

A1: If no failures are detected in round **ρ** of BCA (the round is successful), then at least **nf-f** non-faulty replicas have accepted a proposed transaction in round **ρ**.

A2: If a non-faulty replica accepts a proposed transaction **T** in round **ρ** of BCA, then all other non-faulty replicas that accepted a proposed transaction, accepted **T**.

A3: If a non-faulty replica accepts a transaction **T** , then **T** can be recovered from the state of any subset of **nf-f** non-faulty replicas.

Some things to remember for RCC…

- These checkpoint algorithms run concurrently with the operations of the BCA instances , thereby adhering to our wait-free design goals **D4** and **D5**.

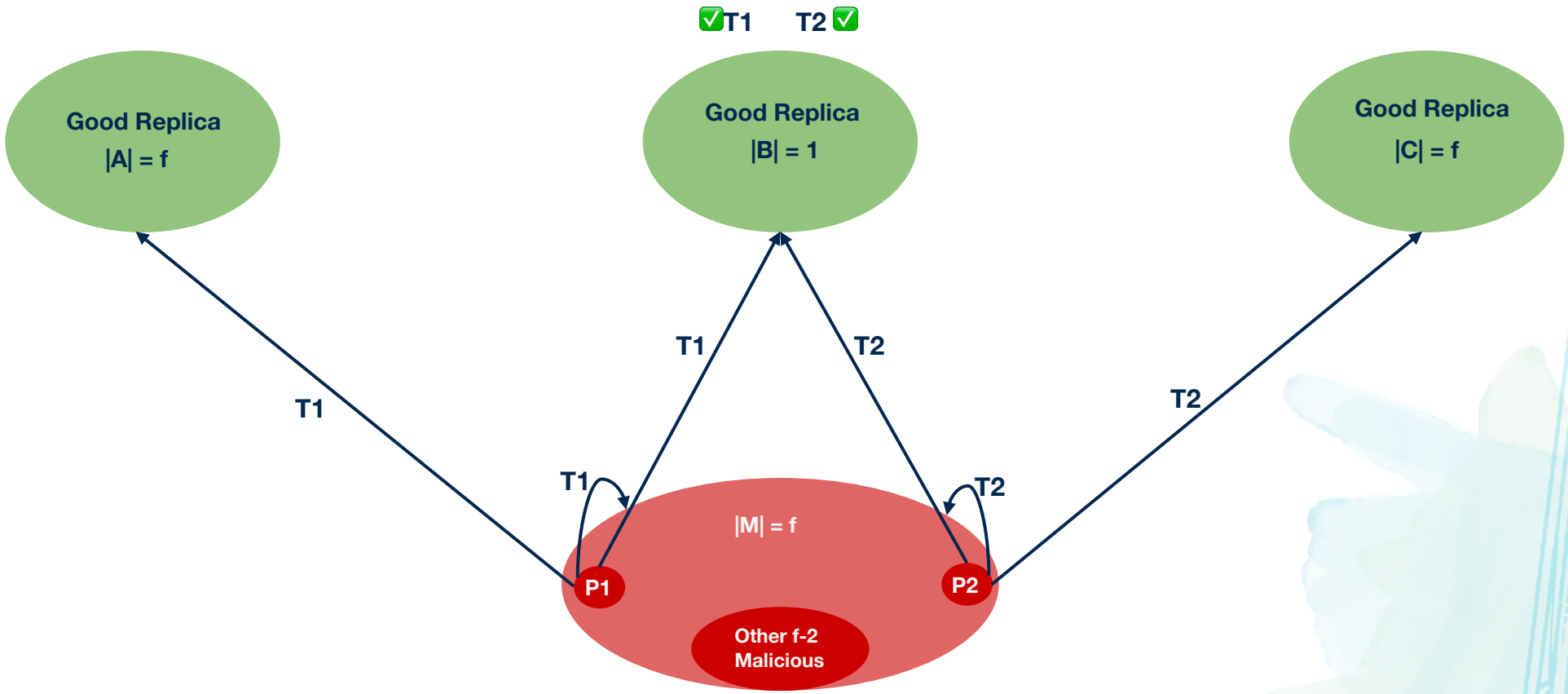
- In RCC, we do this checkpointing on a ***need-basis****: We do it only:*

➔ When Replica R receives **nf - f** claims of failure of **primaries** (via the FAILURE messages of the recovery protocol) in round **ρ**

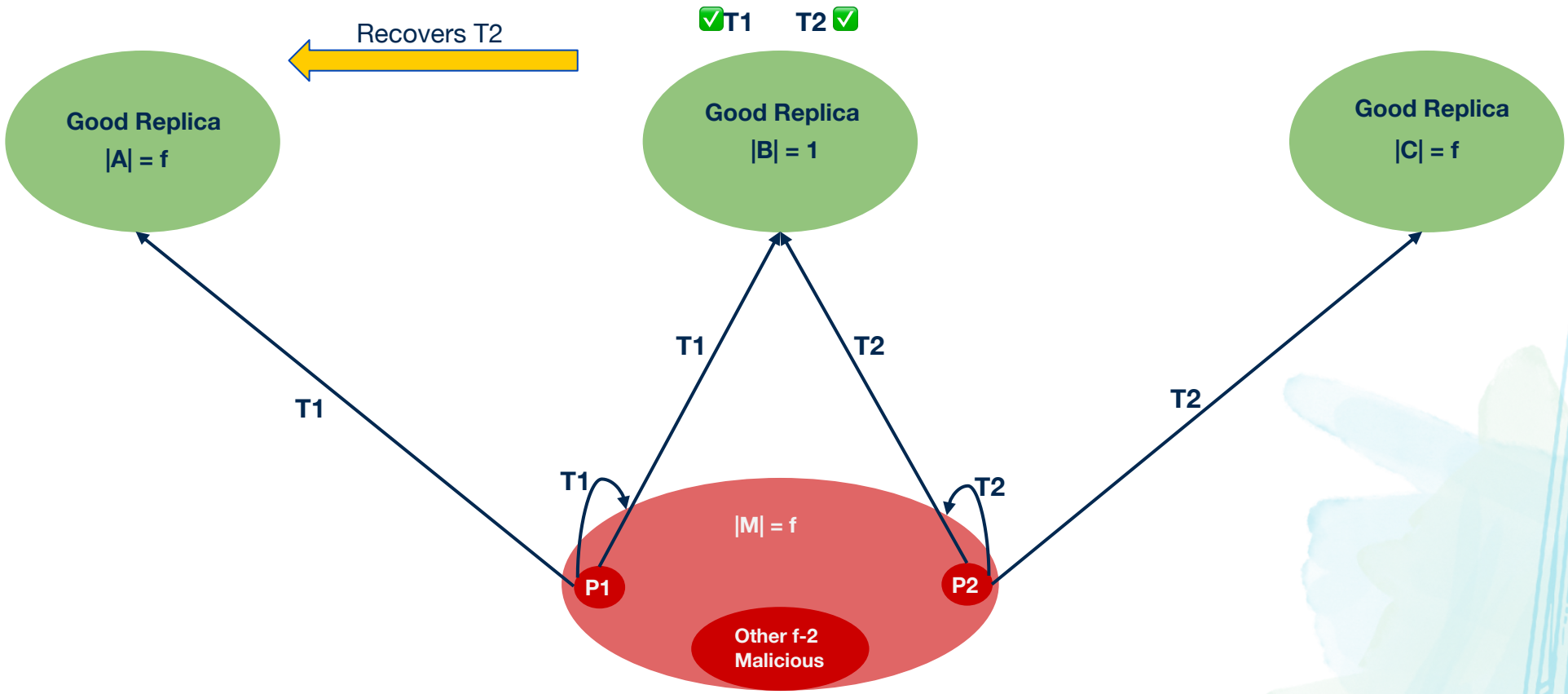
and

➔ **R** itself finished round **ρ** for all its instances ->>only then it will participate in any attempt for a checkpoint for round **ρ.**

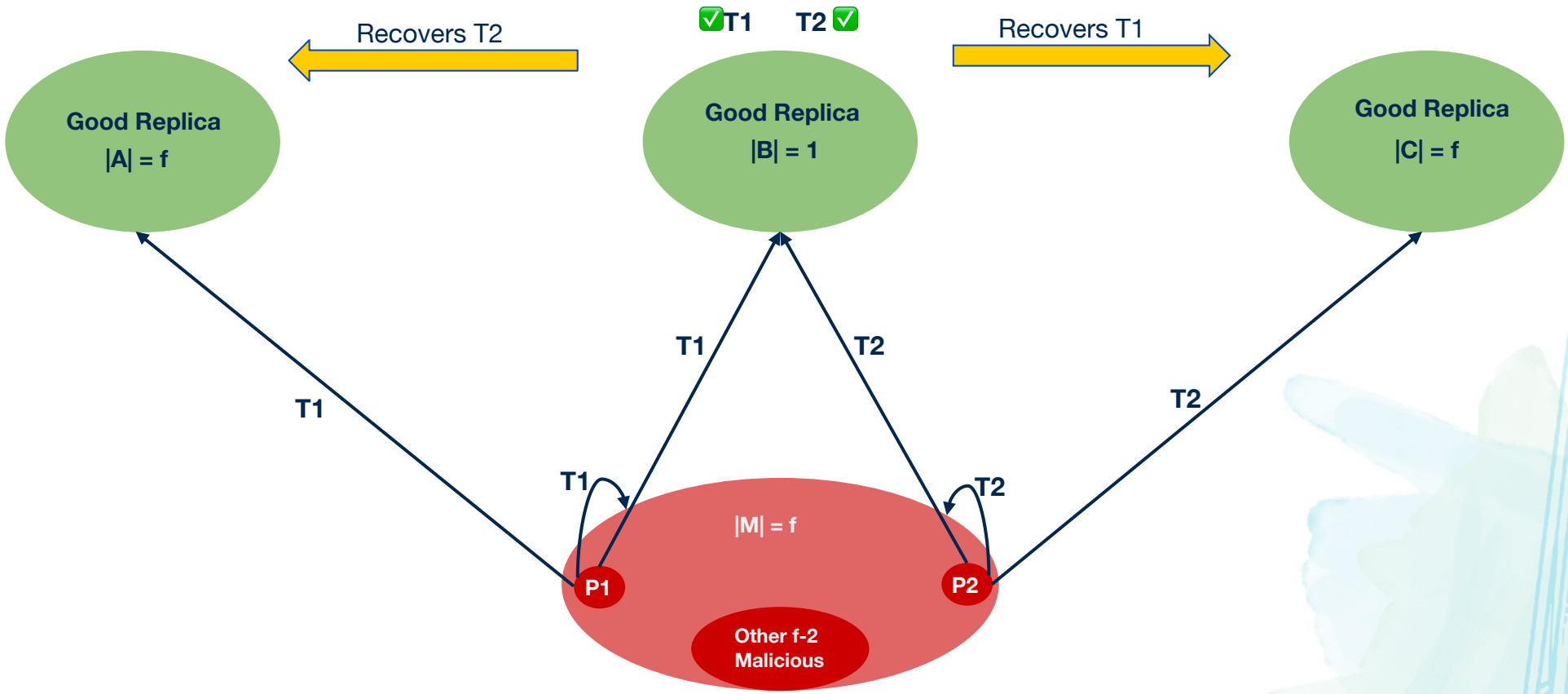
Dealing with Undetectable Failures



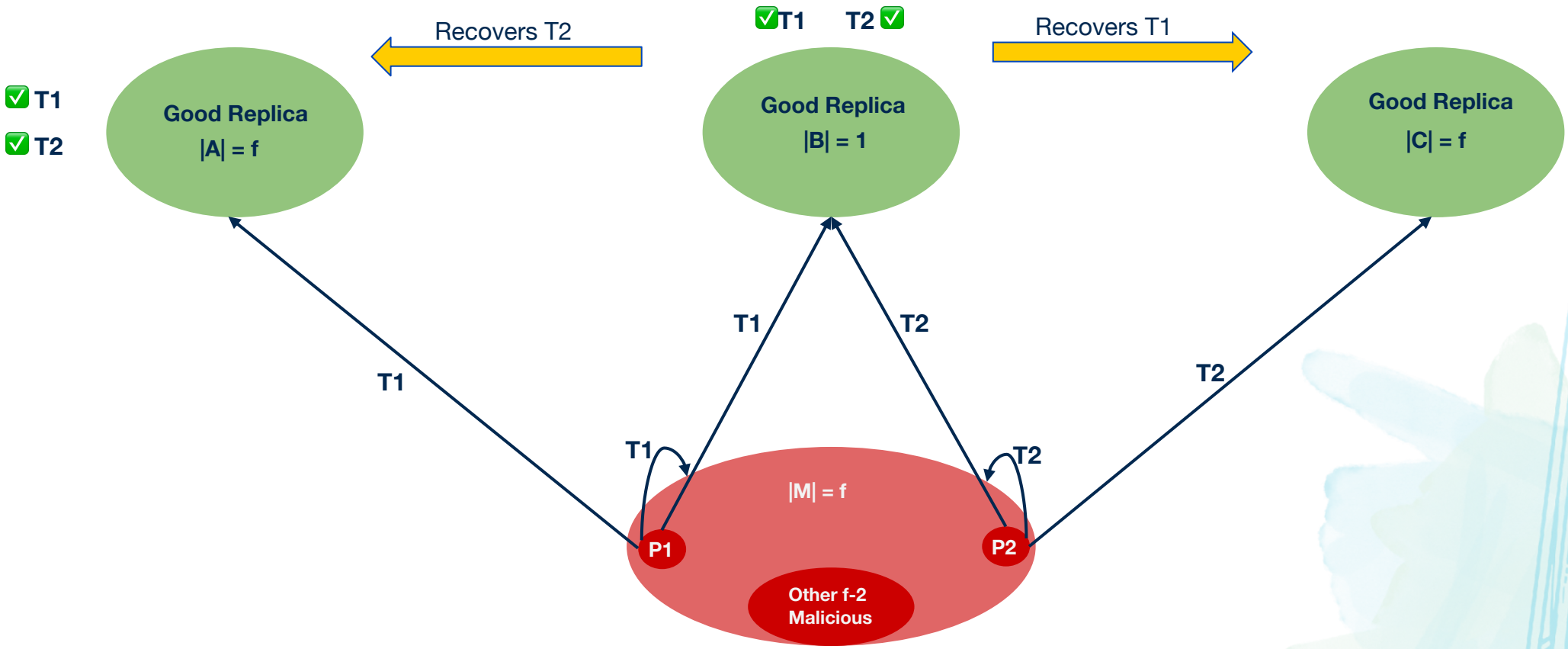
Dealing with Undetectable Failures



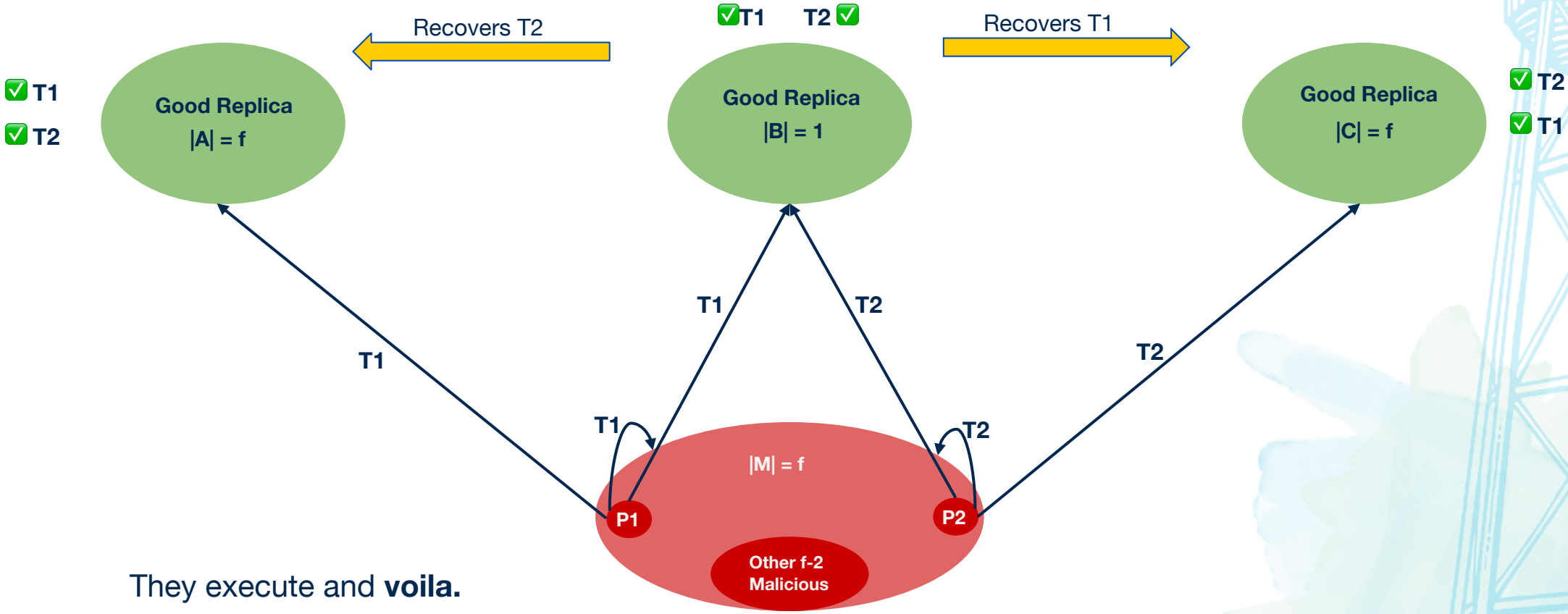
Dealing with Undetectable Failures



Dealing with Undetectable Failures



Dealing with Undetectable Failures



They execute and **voila.**

Client interactions with RCC

- To maximize the throughput, it is very important that each instance proposes distinct client transactions.
- RCC is built to work efficiently in the case where clients >>replicas. It achieves high efficiency by doing the following:
  - Assign each client C_i to a single primary P_i
  - Only P_i propose transactions for C_i

Issue 1: When Clients<Replicas

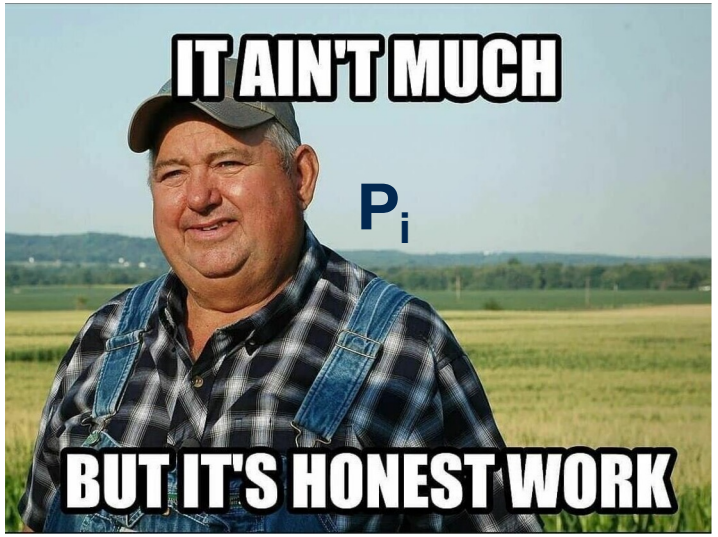
When there are less clients than replicas, some of the replicas need to idle.

Issue 1: When Clients<Replicas

When there are less clients than replicas, some of the replicas need to idle.

Solution!

Any primary P_i proposes a small no-op request when it doesn't have a transaction to propose



Issue 2: When primary doesn't propose

A faulty primary P_i refuses to propose transactions

Issue 2: When primary doesn't propose

A faulty primary P_i refuses to propose transactions

Solution!

1. Proposal is incentivised as P_i will get detected as faulty if it never proposes
2. Client forces execution by broadcasting the transaction to all replicas and replicas detect failure if the primary does not propose this.

Issue 3: When primary can’t propose

A faulty primary P_i refuses to propose transactions

Issue 3: When primary can't propose

A faulty primary P_i refuses to propose transactions

Solution!

C requests to be reassigned by broadcasting SWITCHINSTANCE(c, j)

Improving Resilience of Consensus

- Traditional primary-backup protocols are only designed to deal with primaries that completely fail to propose.
- There are other forms of attacks as well, that they cannot handle.

Ordering attack

Scenario: Bank transaction

T1: transfer(Alice, Bob, 300)

T2: transfer(Bob, Eve, 200)

```
transfer(A, B, M):
        if(bal(A)>m):
        then:
                bal(A)=bal(A)-m
                bal(B)=bal(B)+m
```

Initial state:
Alice: 500
Bob: 0
Eve: 200

Ordering attack

Scenario: Bank transaction

T1: transfer(Alice, Bob, 300)

T2: transfer(Bob, Eve, 200)

```
transfer(A, B, M):
        if(bal(A)>m):
        then:
            bal(A)=bal(A)-m
            bal(B)=bal(B)+m
```

Initial state:
Alice: 500
Bob: 0
Eve: 200

Order1: T1, T1
Alice(300)→Bob ✅
Bob(200)→Eve ✅
Alice: 200
Bob: 100
Eve: 400

Ordering attack

Scenario: Bank transaction

T1: transfer(Alice, Bob, 300)

T2: transfer(Bob, Eve, 200)

```
transfer(A, B, M):
        if(bal(A)>m):
        then:
                bal(A)=bal(A)-m
                bal(B)=bal(B)+m
```

Initial state:
Alice: 500
Bob: 0
Eve: 200

Ordering attack

Scenario: Bank transaction

T1: transfer(Alice, Bob, 300)

T2: transfer(Bob, Eve, 200)

```
transfer(A, B, M):
        if(bal(A)>m):
        then:
            bal(A)=bal(A)-m
            bal(B)=bal(B)+m
```

Initial state:
Alice: 500
Bob: 0
Eve: 200

Order1: T2, T1
Bob(200)→Eve ❌
Alice(300)→Bob ✅
Alice: 200
Bob: 300
Eve: 200

Solution!

- Compute the order of transactions in a deterministic way.
- Order is practically impossible to predict or influence by faulty replicas.
- All replicas arrive at the same order.

Throttling attack

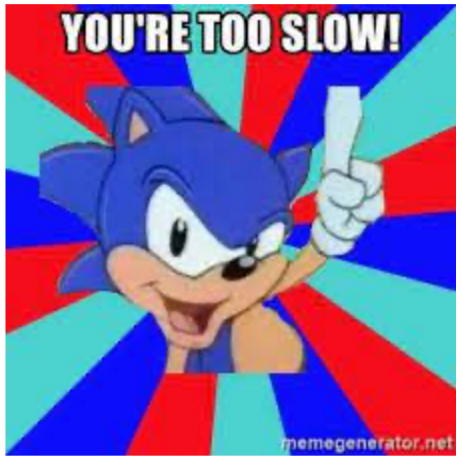
A faulty primary deliberately slow down the system by proposing transactions as slow as possible.

Throttling attack

A faulty primary deliberately slow down the system by proposing transactions as slow as possible.

Solution!

Non-faulty replicas detect the failure of instances that lag behind others by σ rounds



Evaluation: RCC vs Others

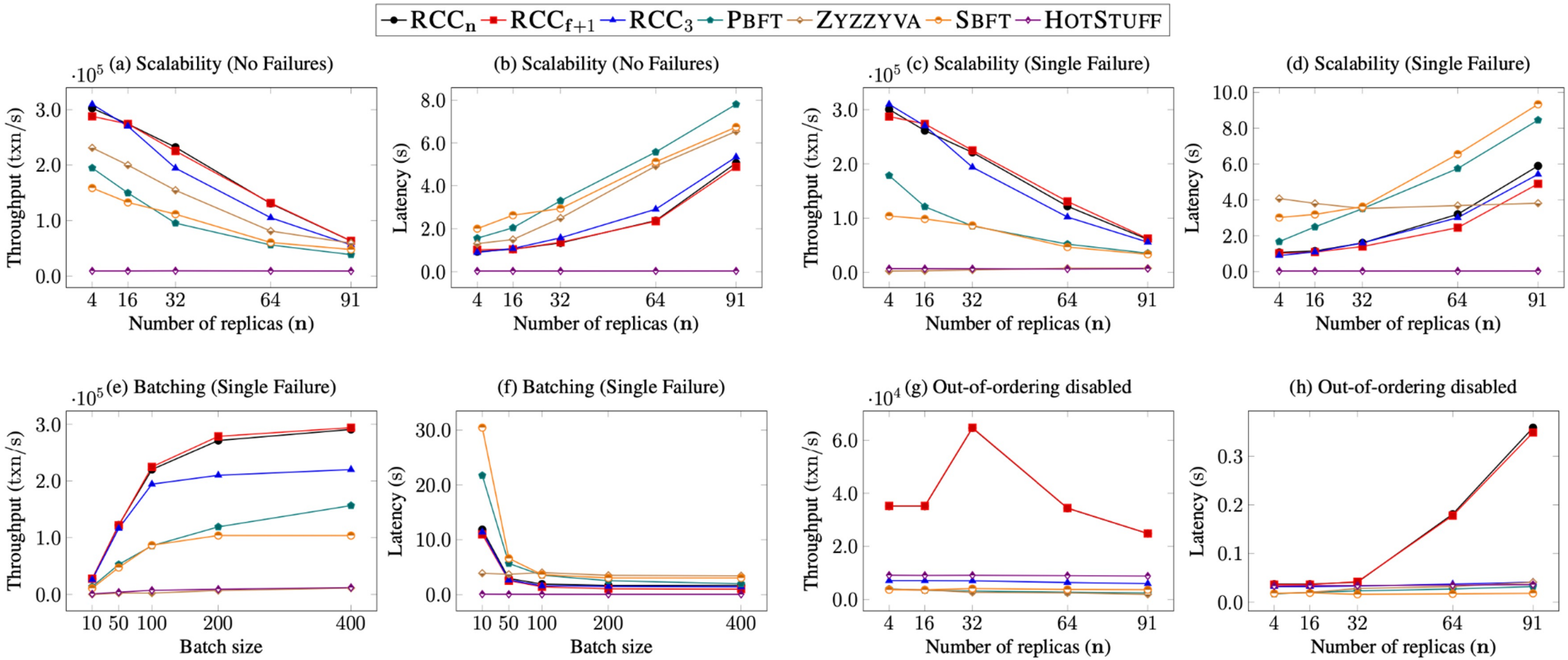
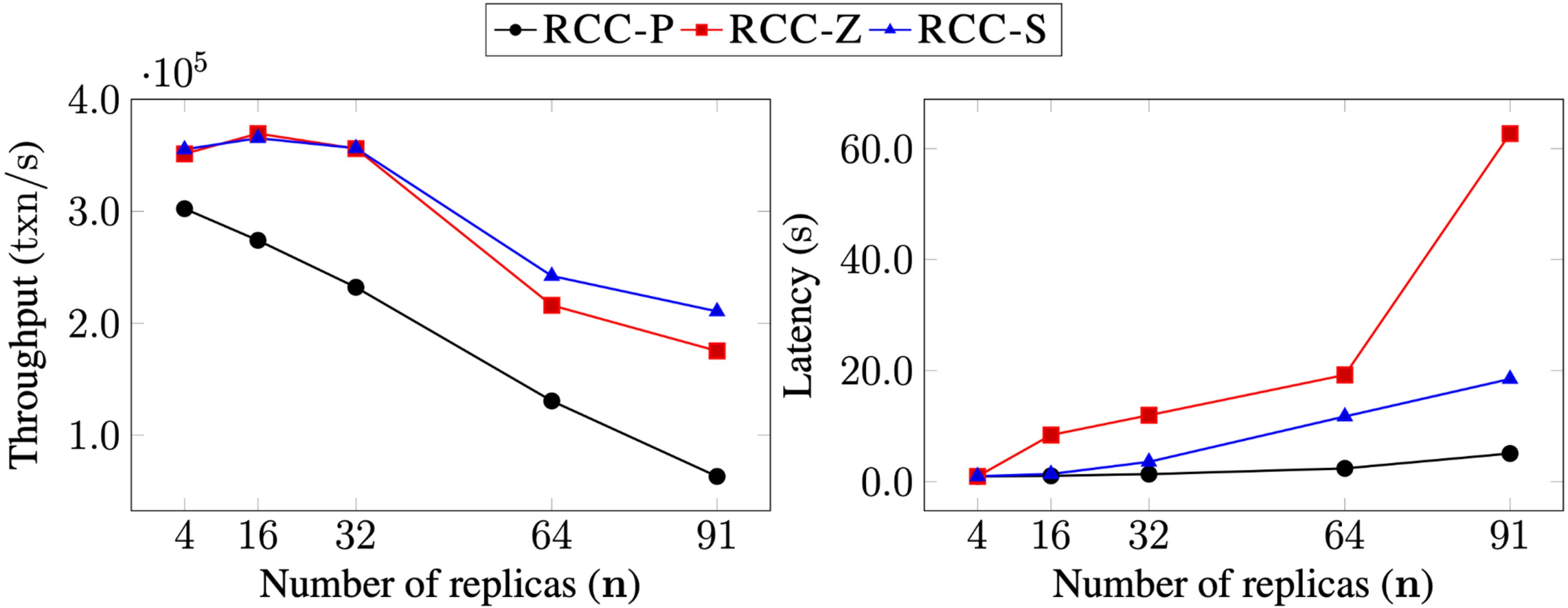


Fig. 8. Evaluating system throughput and average latency incurred by RCC and other consensus protocols.

Evaluation: RCC vs others

- RCC implementations achieve up to 2.77×, 1.53×, 38×, and 82×higher throughput than SBFT, PBFT, HOTSTUFF, and ZYZZYVA in single failure experiments.
- RCC implementations achieve up to 2×, 1.83×, 33×, and 1.45×higher throughput than SBFT, PBFT, HOTSTUFF, and ZYZZYVA in no failure experiments, respectively.

Evaluation: RCC as a paradigm



RCC-Z and RCC-S achieve 3.33x and 2.78x higher throughputs than RCC-P

Conclusion

- A novel concurrent consensus paradigm was established
- Deterministic ordering of transactions
- Protection against both detectable and undetectable faults
- Improved resilience against attacks
- Resulting in improved throughput and reduced latency

References

- S. Gupta, J. Hellings and M. Sadoghi, "RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing," in 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 2021, pp. 1392-1403, doi: 10.1109/ICDE51399.2021.00124.
- Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In Proceedings of the third symposium on Operating systems design and implementation (OSDI '99). USENIX Association, USA, 173–186.


THANK YOU
BEARY MUCH!