Cocytus:

Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication

USENIX FAST ' 16

Heng Zhang, Mingkai Dong, Haibo Chen

In-memory KV-Store

A key building block for many systems

- Data cache (e.g., Memcached in Facebook)
- In-memory database (e.g., Redis)

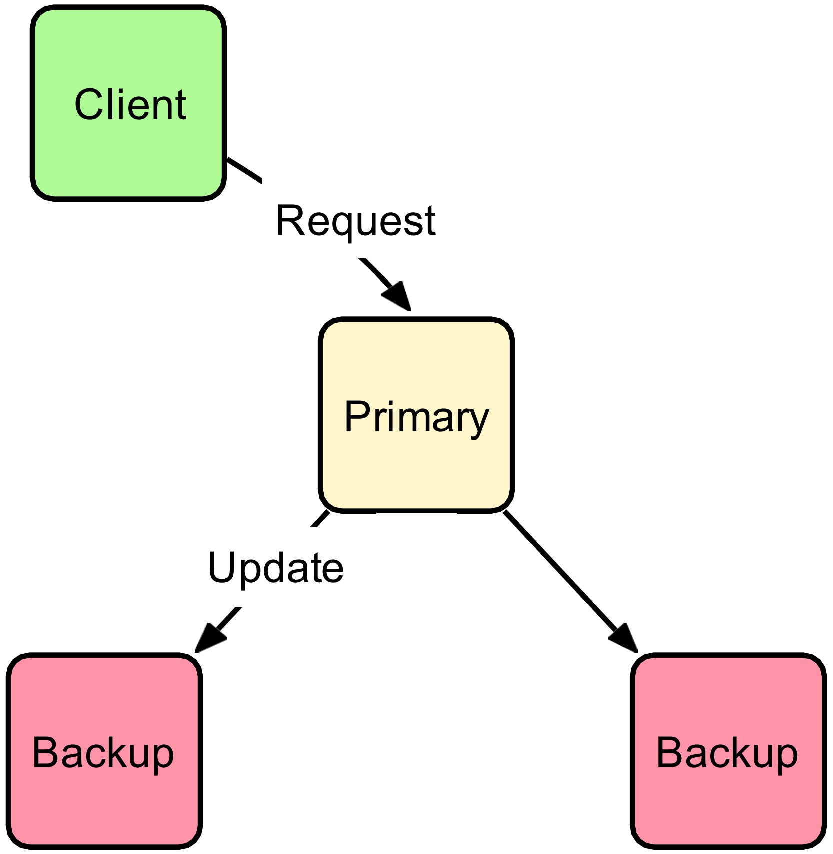
requires availability and efficiency

Primary Backup Replication(PBR)

A common way to achieve availability

- Redis, Repcached

How to improve storage efficiency ?

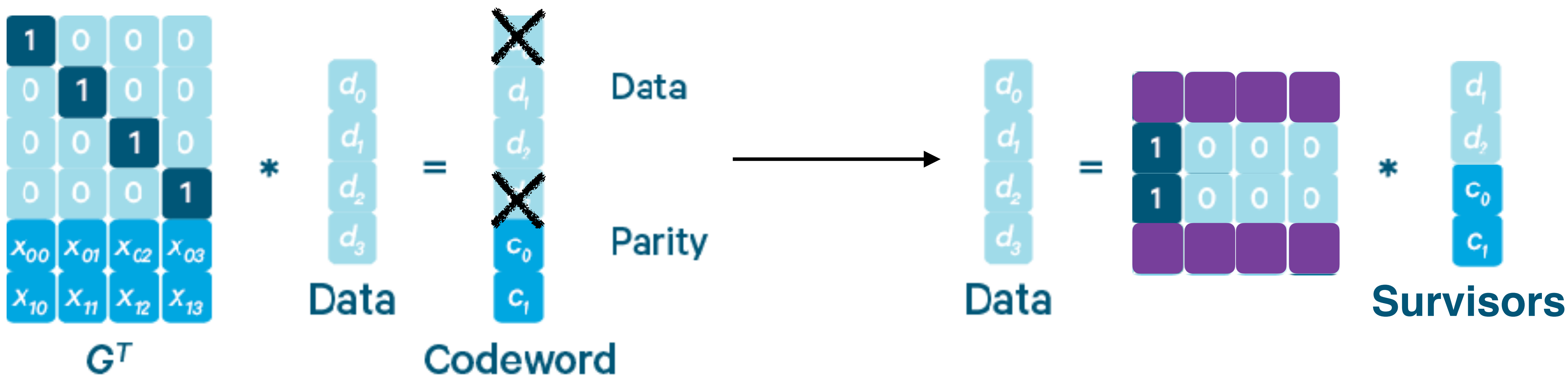


**storage efficiency: 33%*

Erasure Coding(EC)

Reed solomon

- RS(k,m): k data block, m parity block, tolerant m failure



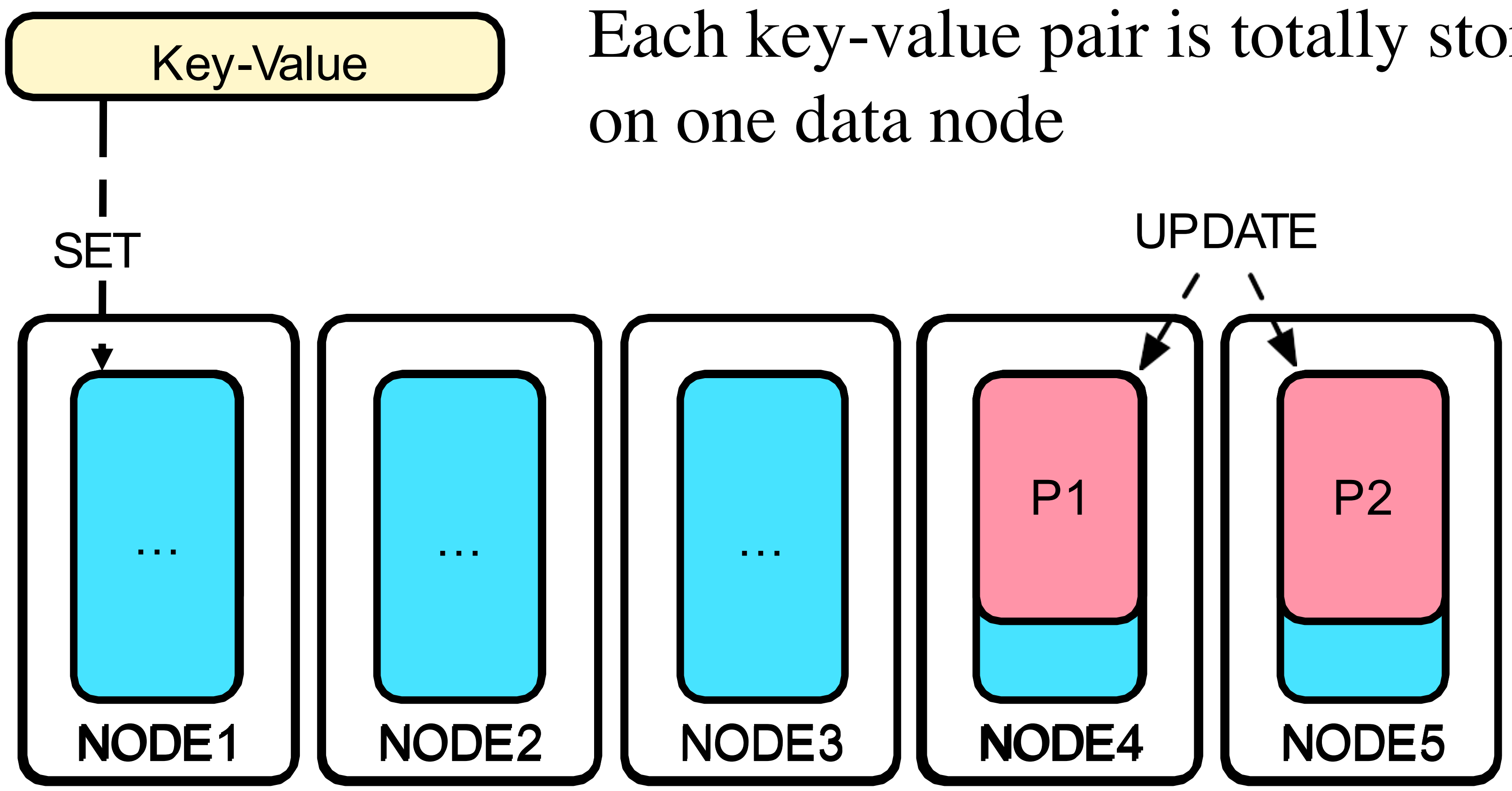
* *GT is a generator matrix*

EC is a space-efficient solution to prevent data loss

Erasure Coding + In-memory KV-Stores

$\xrightarrow{?}$ **Available and Memory Efficient**
In-memory KV-Stores

Intuited System Design



Each key-value pair is totally stored on one data node

K Nodes for storing data   M nodes for storing parity

Challenges

- Excessive metadata update
- Online recovery

Challenge #1: Excessive Metadata Update

- Metadata is usually achieved by scattered and linked data structure (e.g., hash table, binary search tree)

  → **Complicated implementation**

- Operations on metadata involve many scattered modifications (e.g., O(N) modifications on resizing of hash table)

  → **Encoding/transfer consuming**

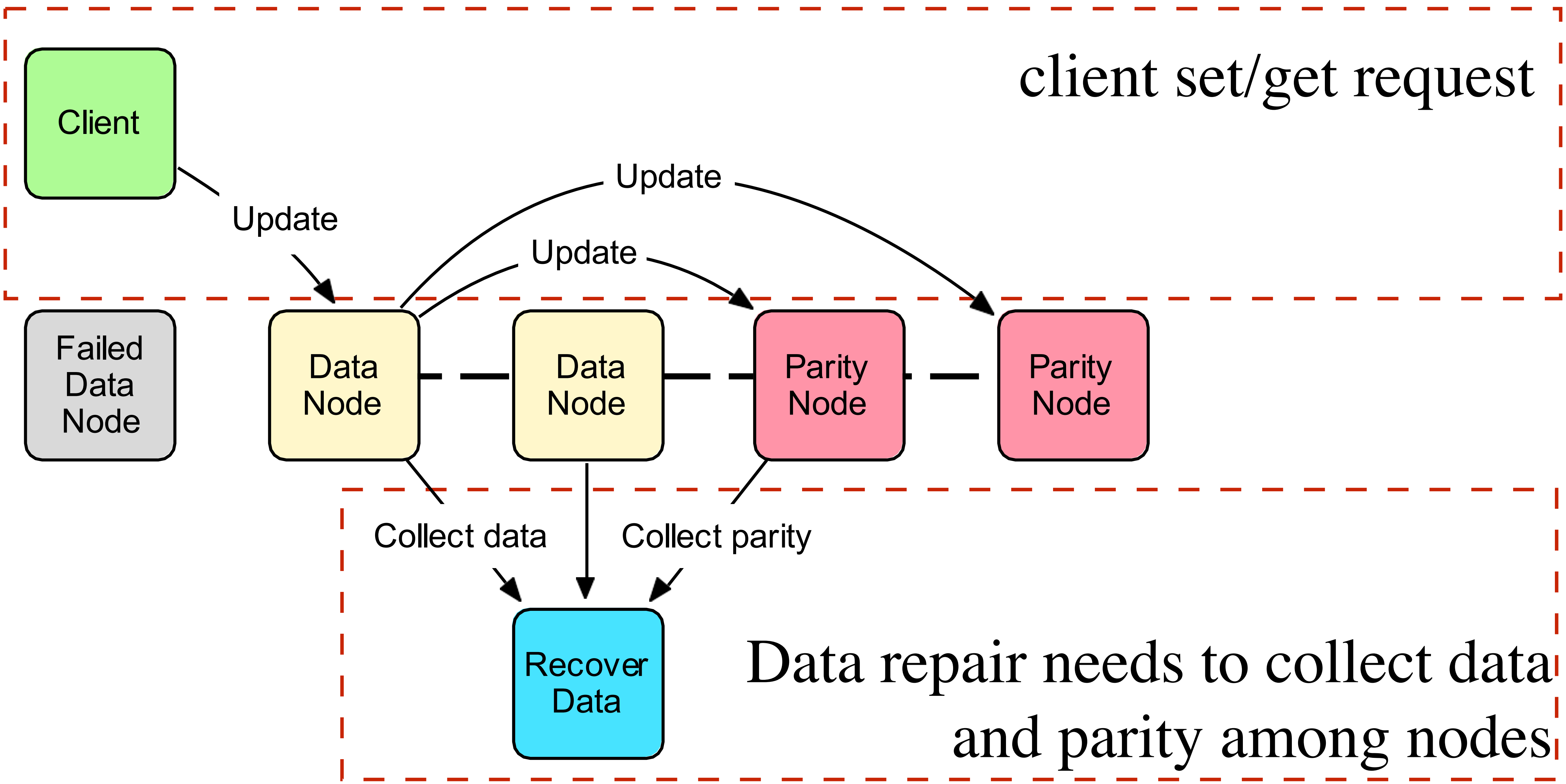
Thus, erasure coding is not a good choice for metadata

Solution: Separate data and metadata

- Use **erasure coding** to prevent **data (values)** loss
  - Pre-allocate virtual memory areas for data and parity
  - Modifications on these areas agree with erasure coding approach

- Use **primary-backup replication** to prevent **metadata** loss
  - Mapping information and allocation information are placed on outside of the area

Challenge #2: Race Condition on Online Recovery



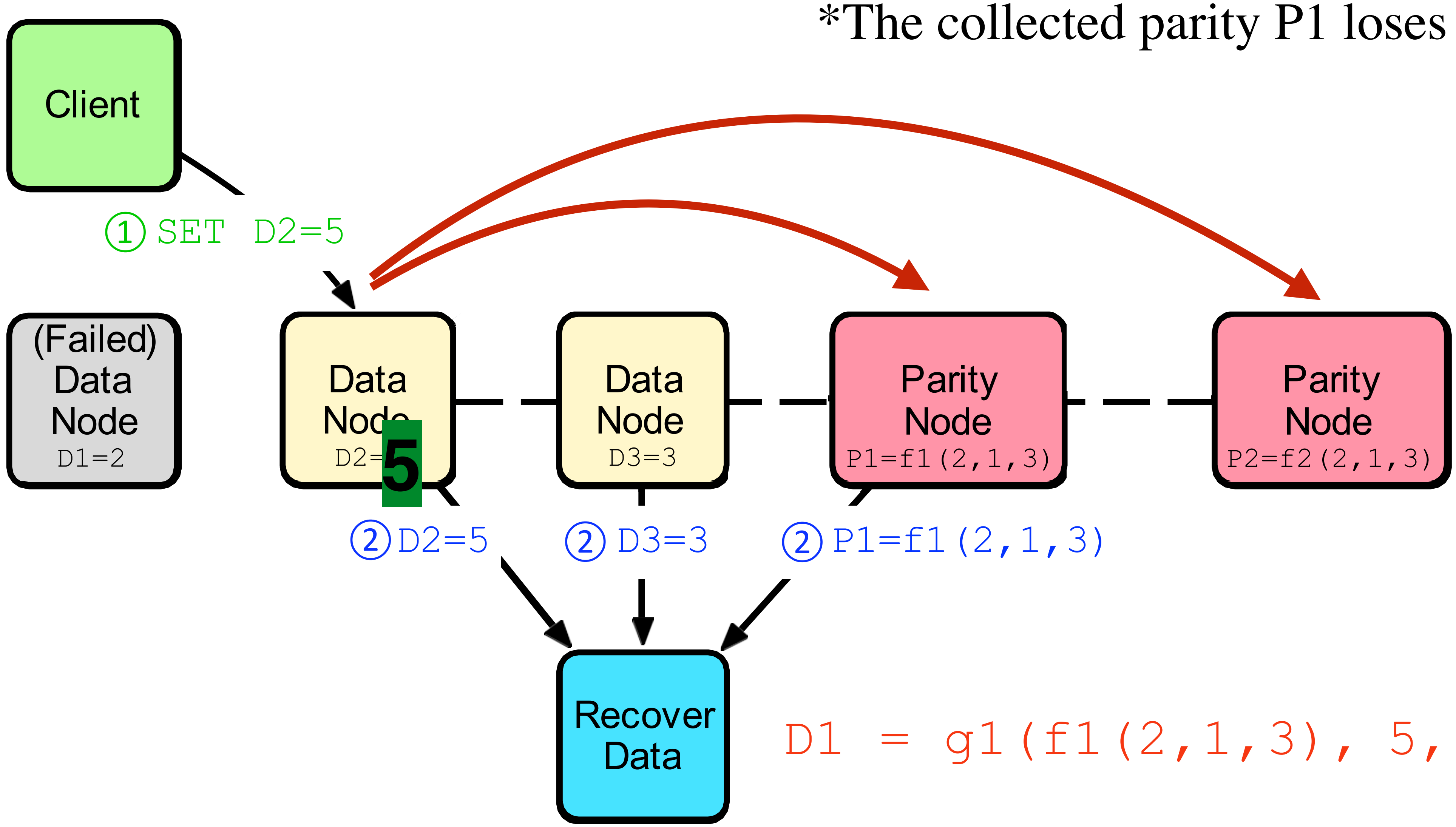
Challenge #2: Race Condition on Online Recovery

- The interleaving of SET requests and data repair has race condition

*f1() and f2() are the encoding function

*g1() is a decoding function

*The collected parity P1 loses the new update

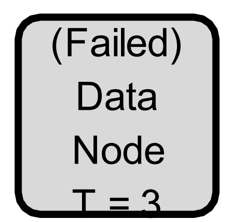


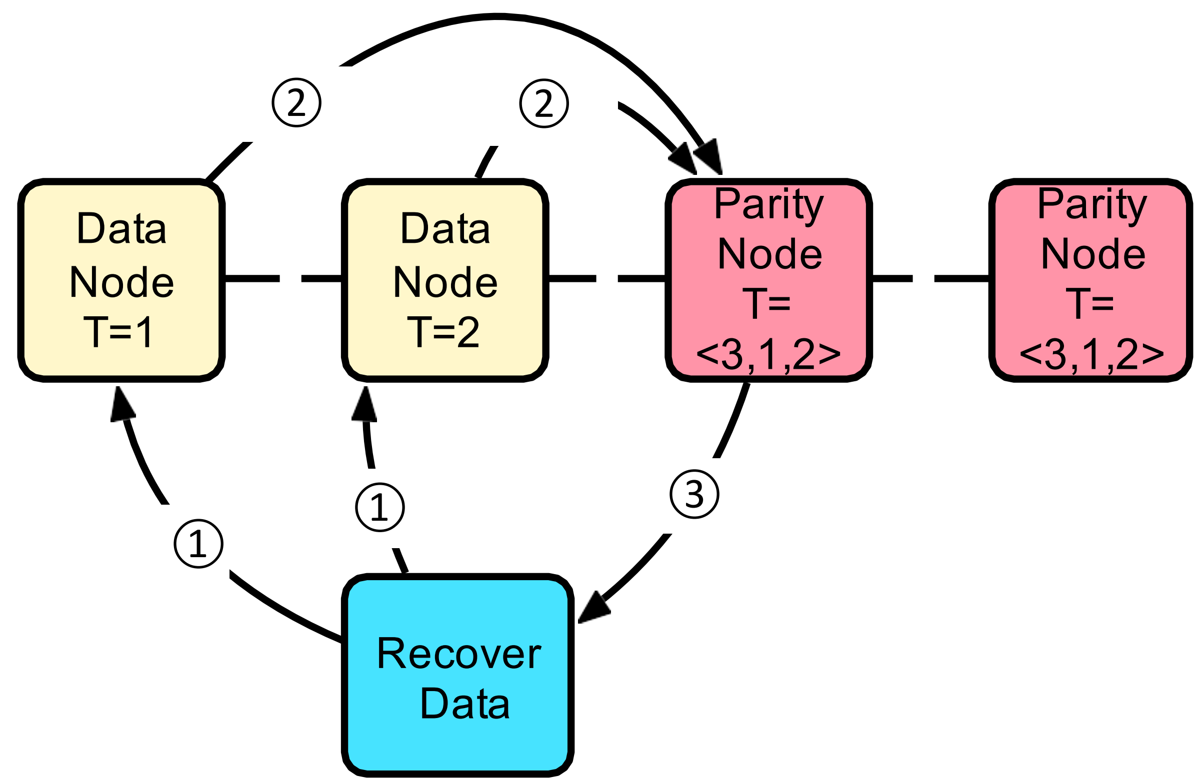
D1 = g1(f1(2,1,3), 5, 3) != 2

Solution: Online Recovery Protocol

- Use **logical timestamp** to indicate the version of data
  - Attach timestamps on SET requests
  - In-order completion

- Three steps for data collection
  - Start procedure
  - Decide data versions
  - Synchronize parity version

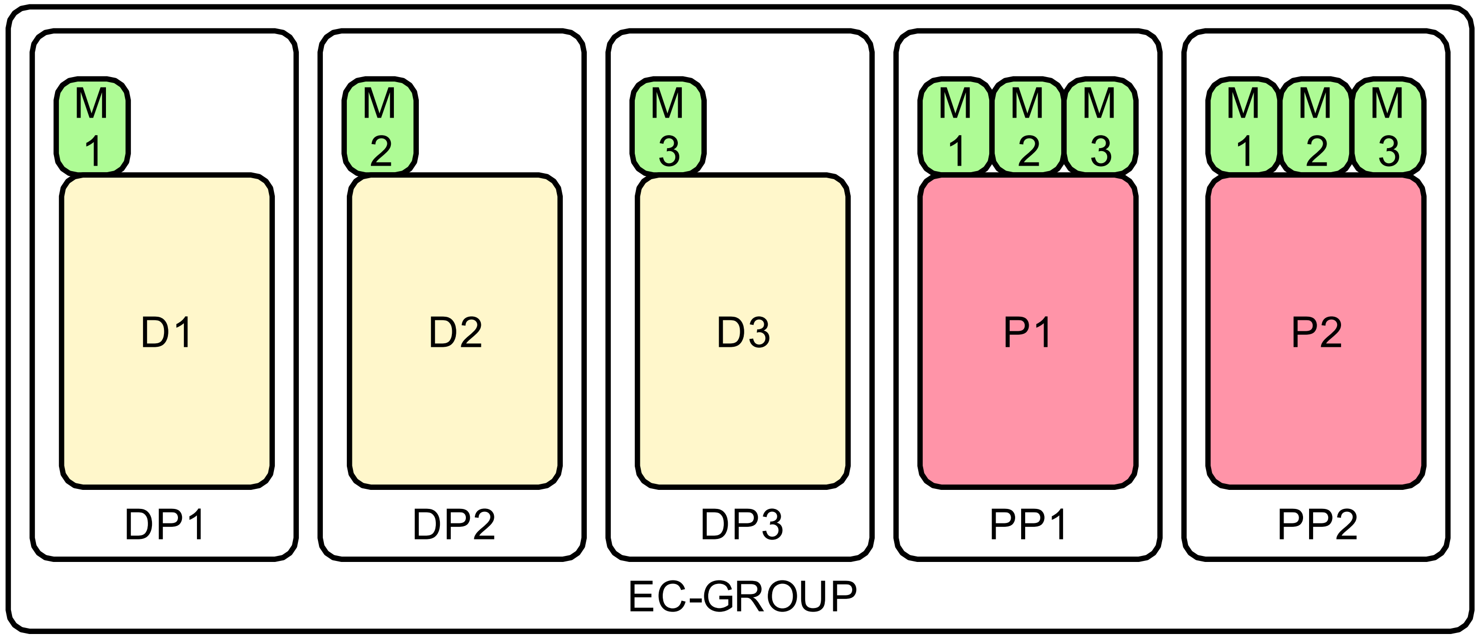




Overview

Cocytus Overview

- EC-Group is the basic component in Cocytus
  - A EC-Group consists K data processes and M parity processes
  - Connected by a FIFO channel like a TCP connection

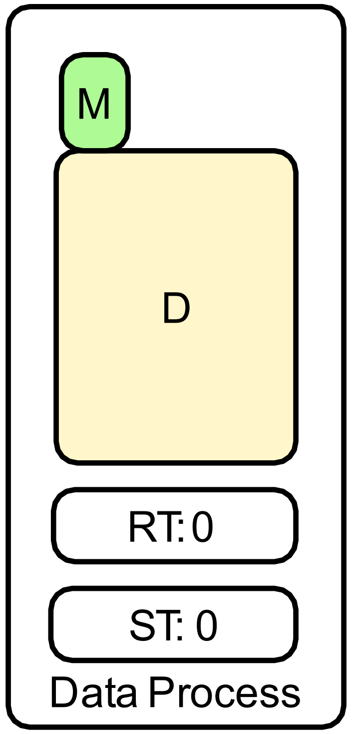


Data Process

- Metadata
  - Mapping information
  - Allocation information

- Data area
  - A memory area for values

- Logical Timestamp
  - A **T**imestamp for the latest **R**eceived SET request (RT)
  - A **T**imestamp for the latest **S**table (completed) SET request (ST)

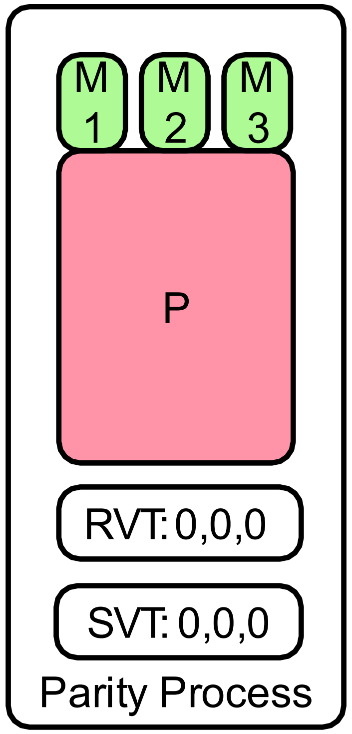


Parity Process

- Metadata replicas of all data processes in the EC-Group

- Parity area
  - A memory area for parity

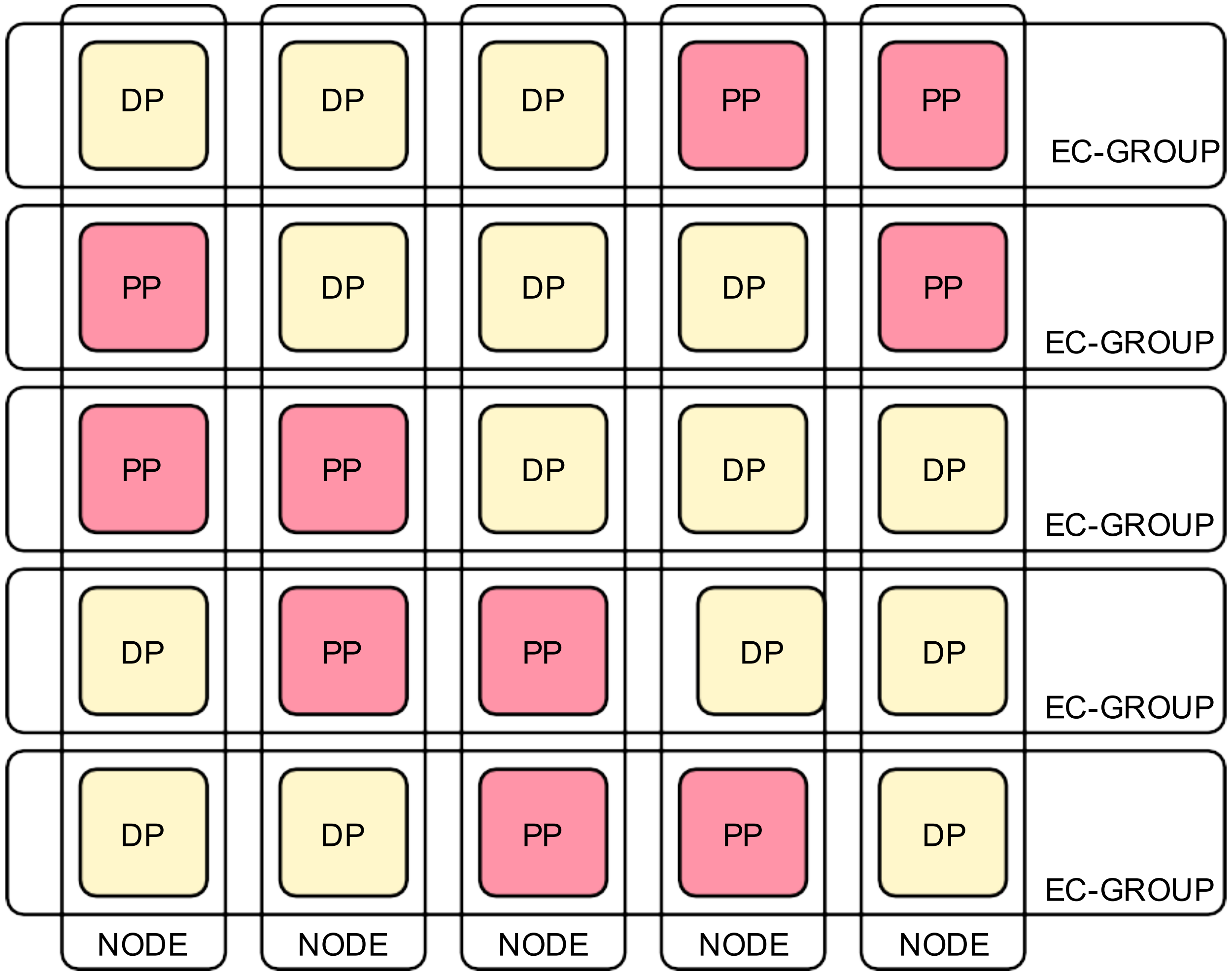
- Logical Timestamp
  - A **T**imestamp **V**ector for the latest **R**eceived SET request (RVT[1…K])
  - A **T**imestamp **V**ector for the latest **S**table (completed) SET request (SVT[1…K])



Layout

- Parity processes save more metadata than data process
- Parity processes only need to participate in *set* operations

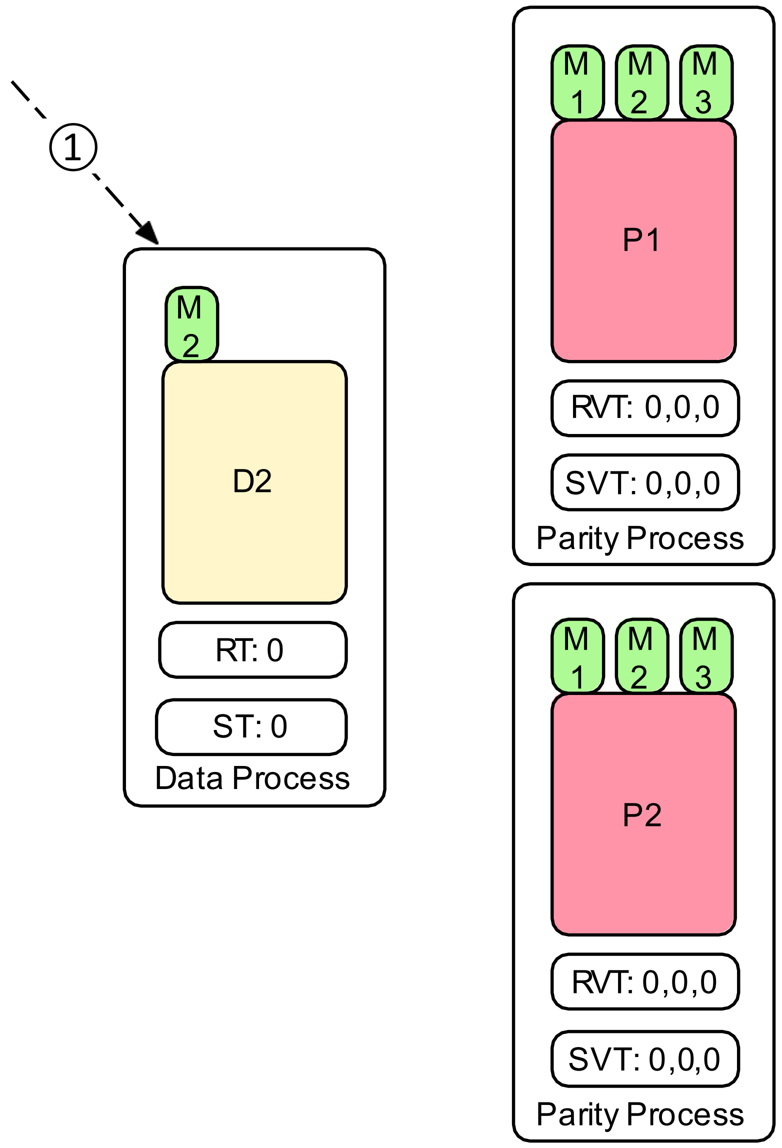
interleaved layout



Handling SET Request

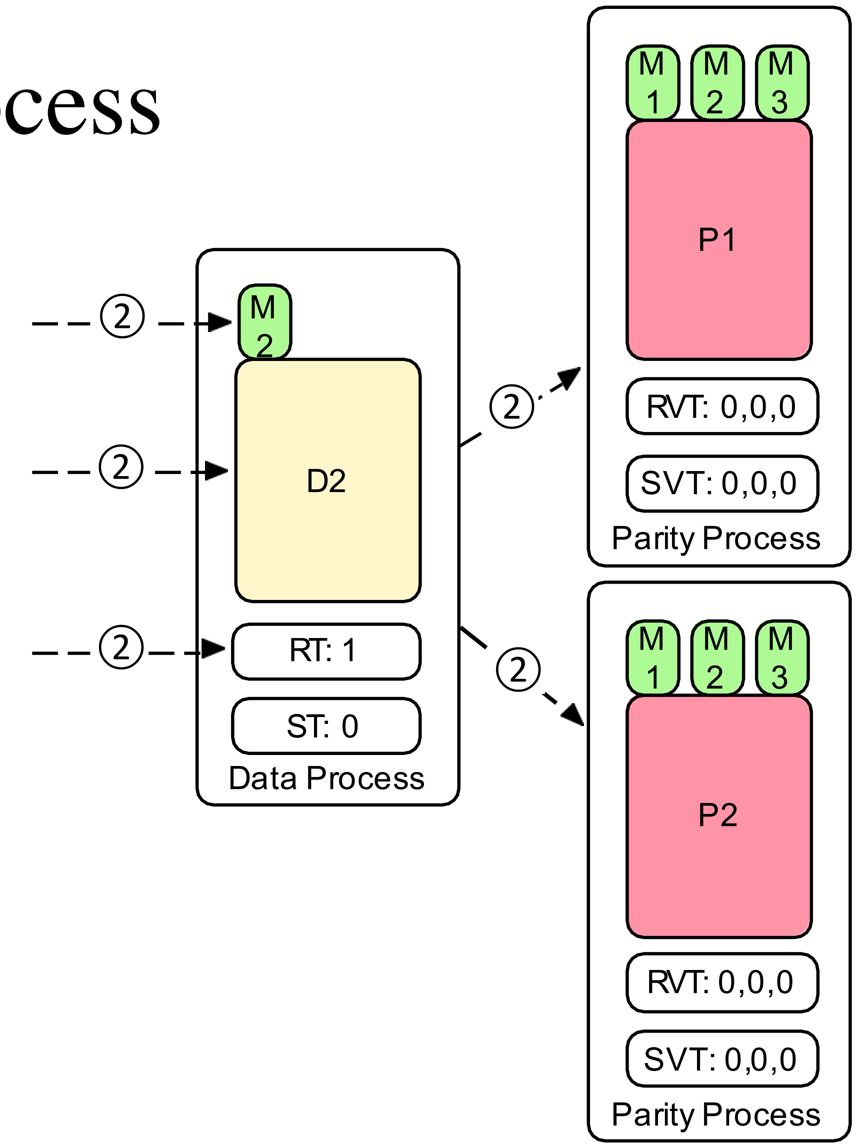
Handling SET Requests

- Dispatch a data process



Handling SET Requests

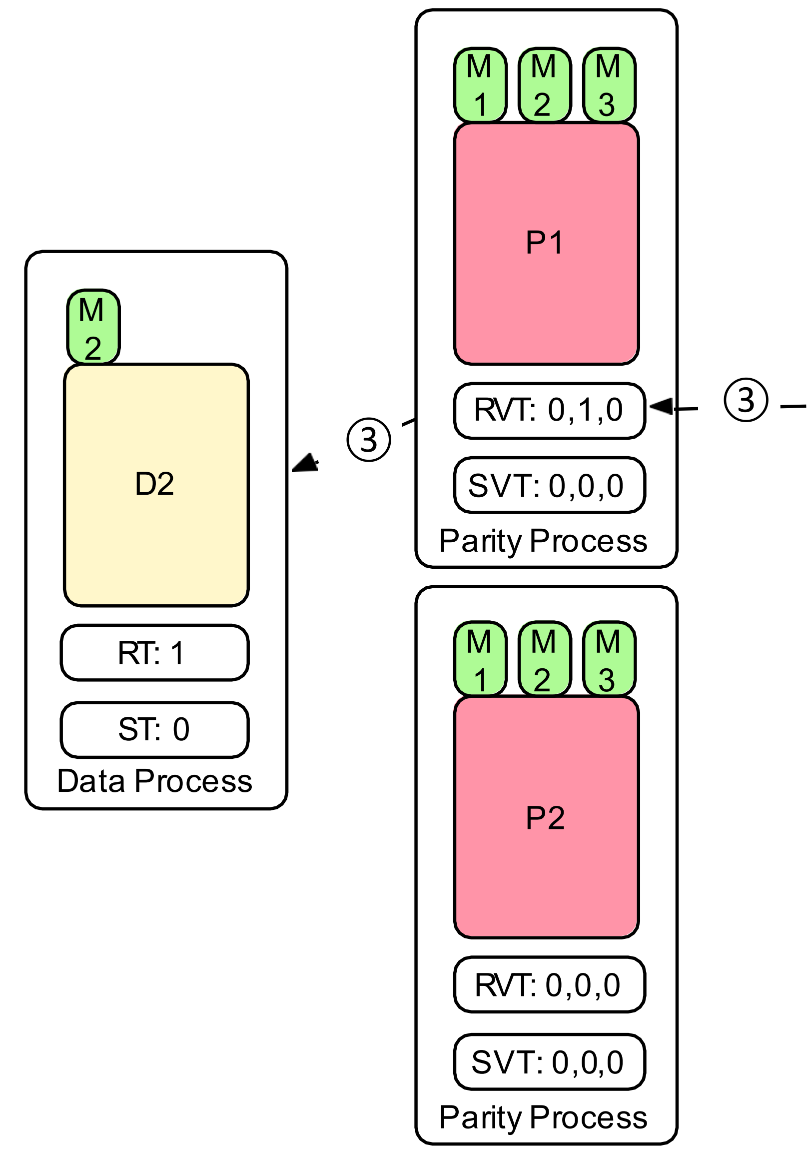
1. Dispatch a data process
2. Handle the request on the data process
    1. Generate data diff
    2. Update the timestamp RT
    3. Forward request



* A **T**imestamp for the latest **R**eceived SET request (RT)

Handling SET Requests

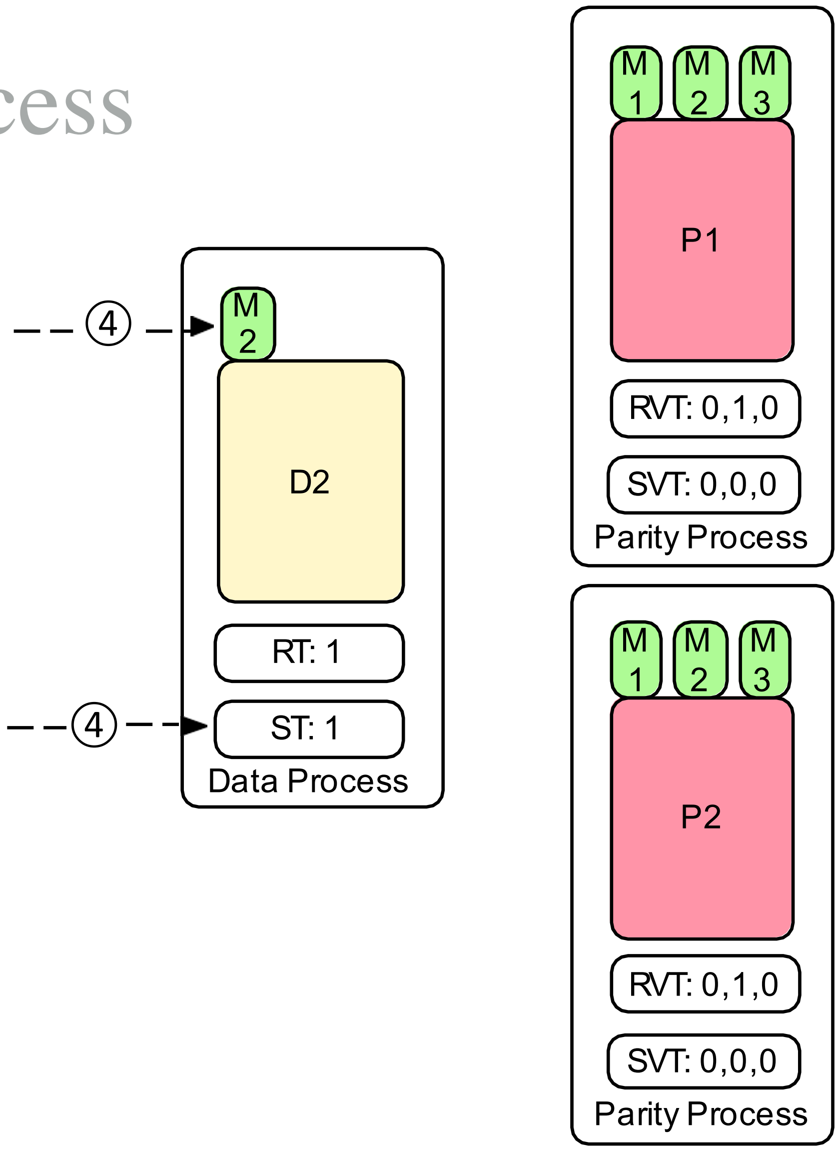
1. Dispatch a data process
2. Handle the request on the data process
3. Handle the request on the parity process
    1. Buffer the request
    2. Update the timestamp RVT
    3. Send ACKs



* A **T**imestamp **V**ector for the latest **R**eceived SET request (RVT[1…K])

Handling SET Requests

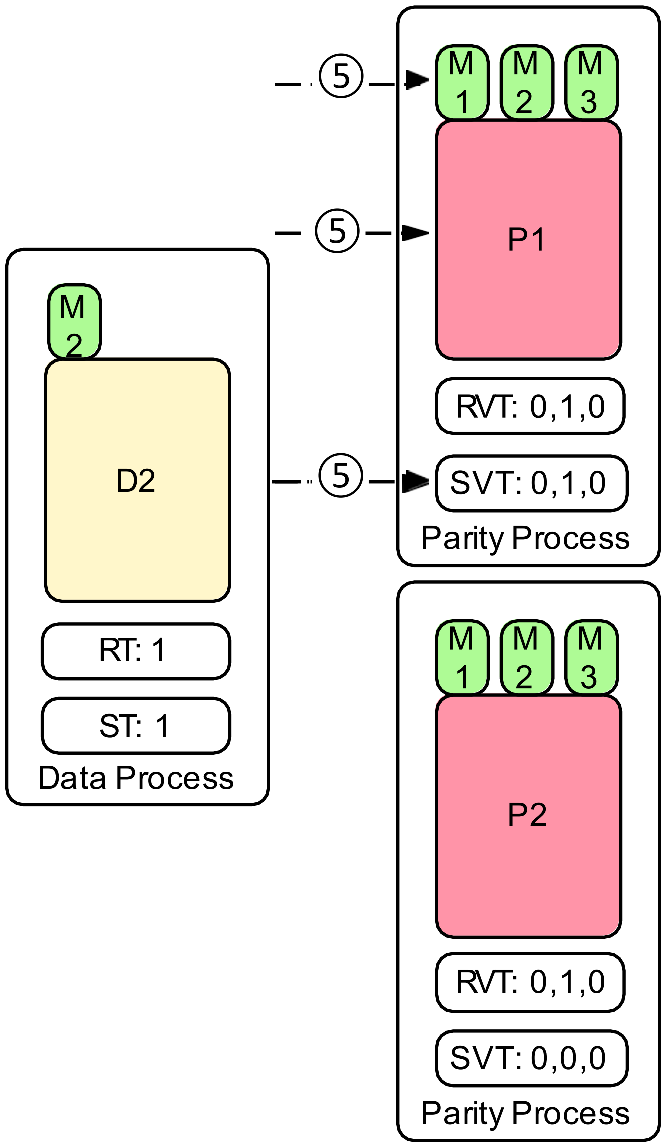
1. Dispatch a data process
2. Handle the request on the data process
3. Handle the request on the parity process
4. Complete the request on the data process
    1. Update in place
    2. Update the timestamp ST
    3. Send commit requests



* A **T**imestamp for the latest **S**table (completed) SET request (ST)

Handling SET Requests

1. Dispatch a data process
2. Handle the request on the data process
3. Handle the request on the parity process
4. Complete the request on the data process
5. Complete the request on the parity process
    1. Update corresponding metadata
    2. Update parity area with diff
    3. Update SVT



* A **T**imestamp **V**ector for the latest **S**table (completed) SET request (SVT[1…K])

Online Recovery

Online Recovery

When a data process fail, Cocytus chooses a recovery leader from parity processes

- Provide continuously services
- Start two-phases recovery
    - Preparation
    - Online data repair

Preparation

The recovery process synchronizes stable timestamp for the failed data process

1. collect corresponding RVT[i]s from all parity processes, where i is the failed data node
2. choose the minimal one to be the synchronized stable

After preparation phase, all parity processes are consistent in the failed data process

Parity processes complete the buffered requests that

- contain equal or smaller timestamps than the synchronized stable timestamp
- come from the failed data processes

Online Data Repair

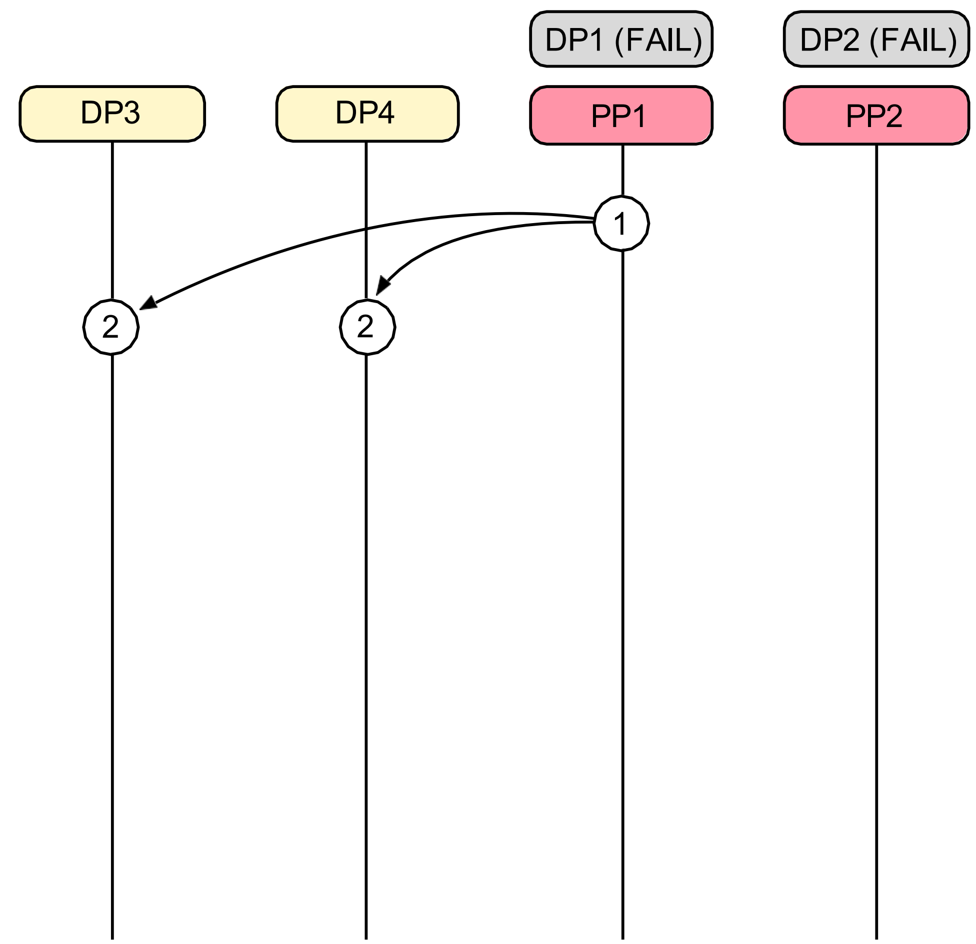
Data area is repaired in a granularity of 4KB page

Under online recovery protocol

Recovery Protocol

Recovery Leader

1. Choose the parity participant
2. Notify alive data processes



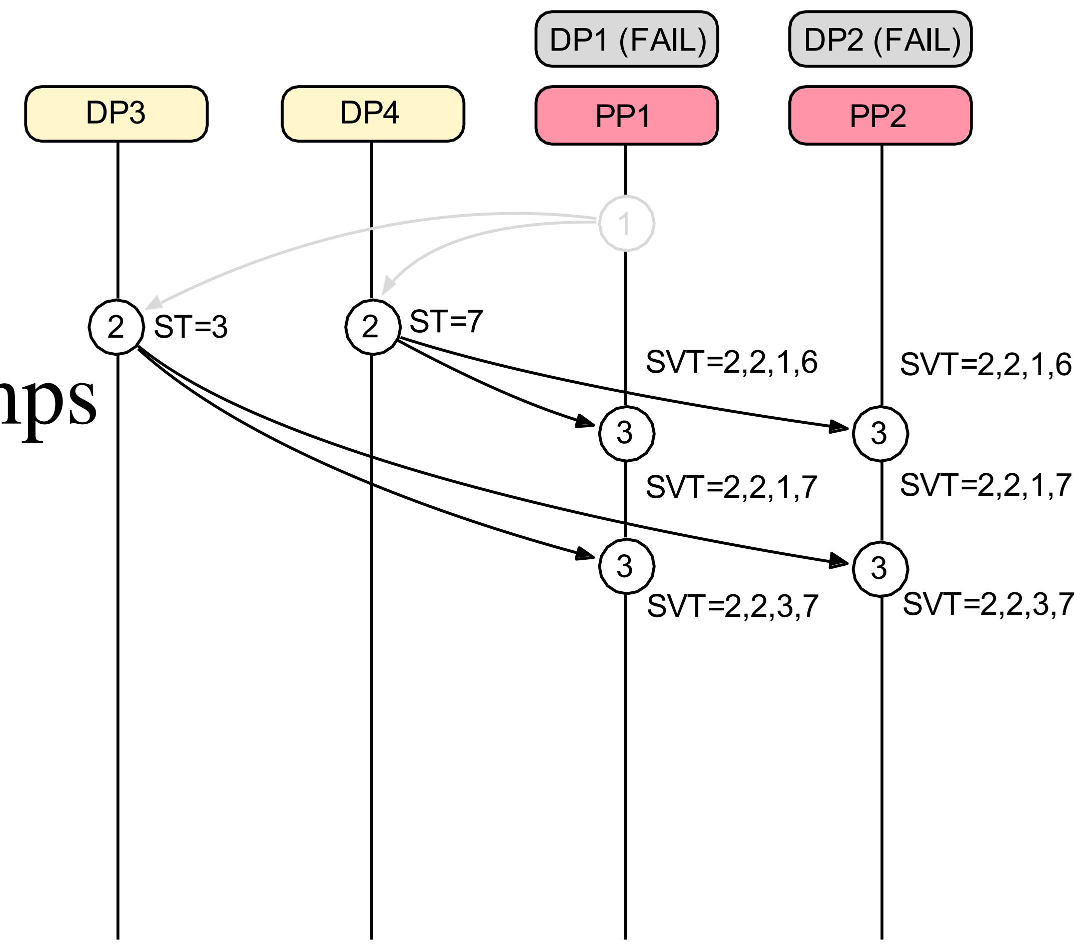
Recovery Protocol

Data Processes

1. Decide stable timestamp
2. Send data page

Recovery Processes

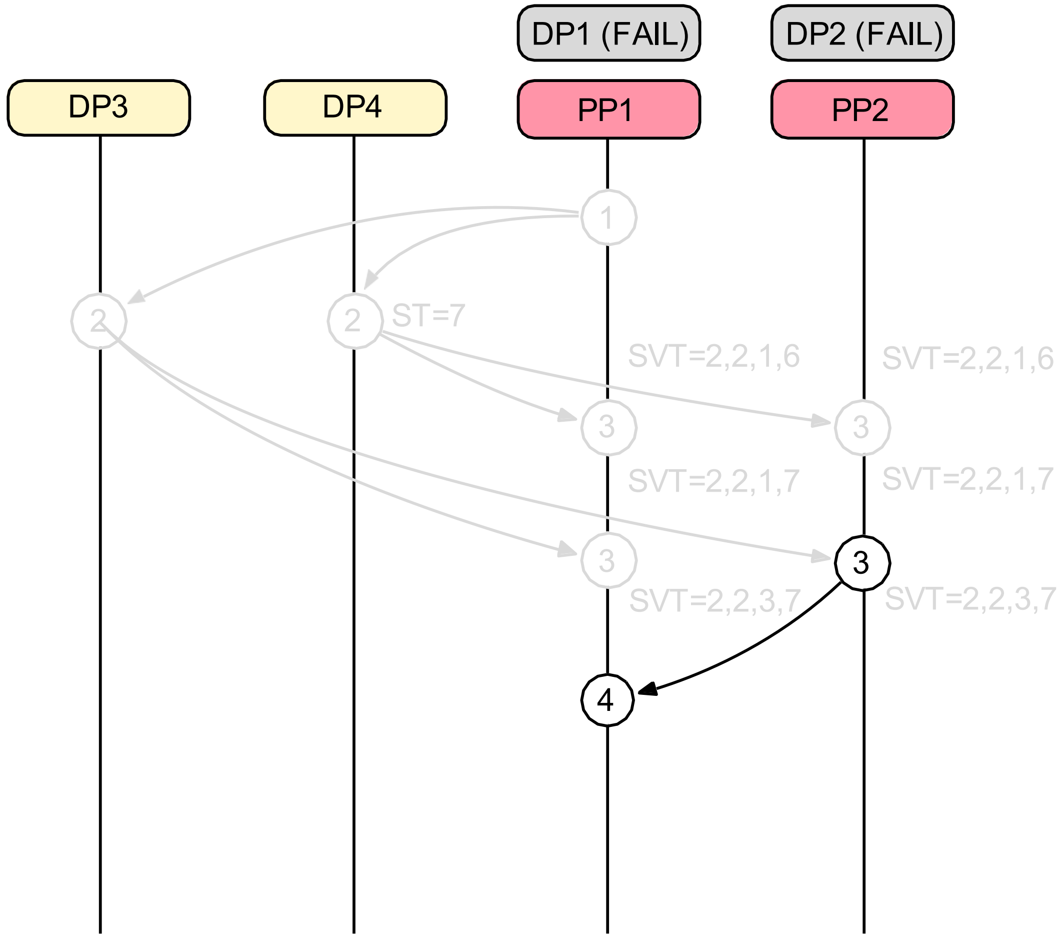
1. Synchronize the stable timestamps
2. Do partial decoding



Recovery Protocol

Recovery Processes

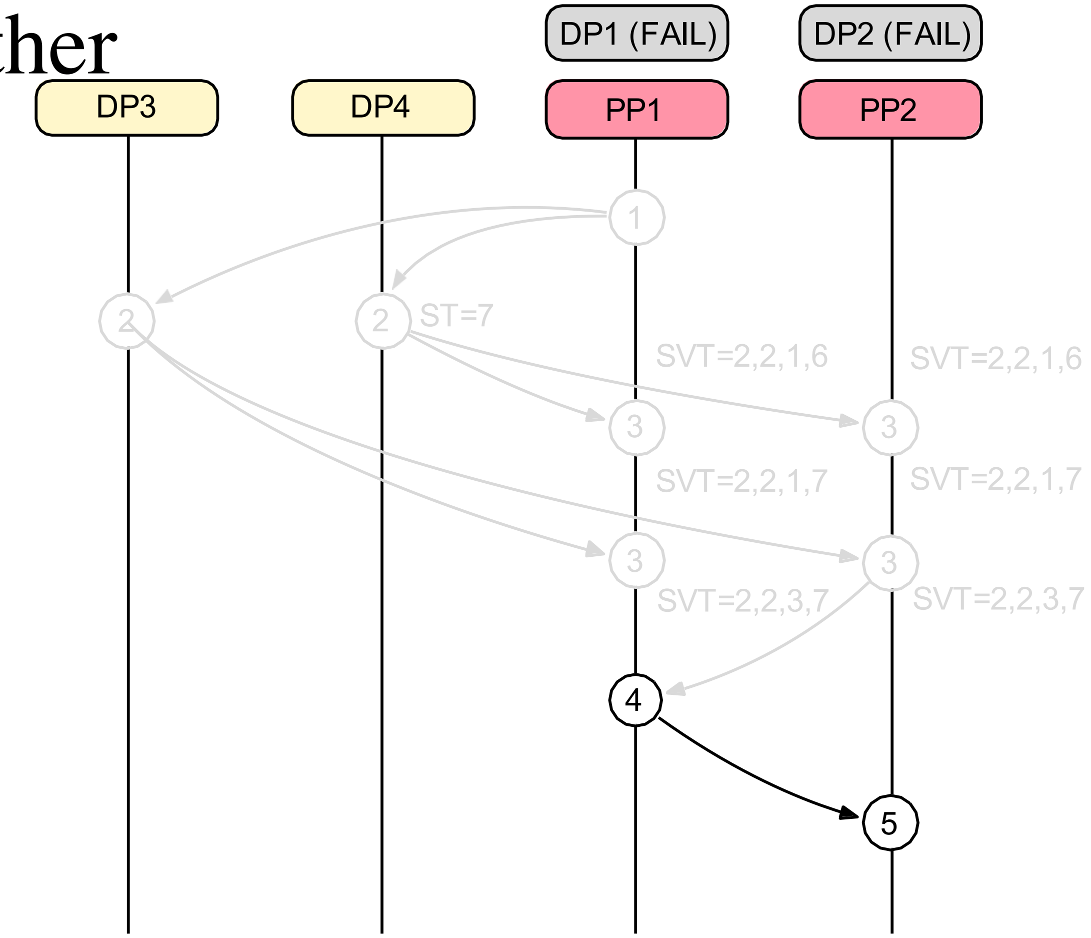
1. send partially decoded parity



Recovery Protocol

Recovery Leader

1. Complete the decoding
2. Send recovered data pages to other recovery processes



Implementation

Cocytus is implemented on Memcached 1.4.21

- Implement a similar primary-backup replication version for comparison

Coding Scheme

- Reed-Solomon code provided by Jerasure

Performance Evaluation

5-node cluster for server

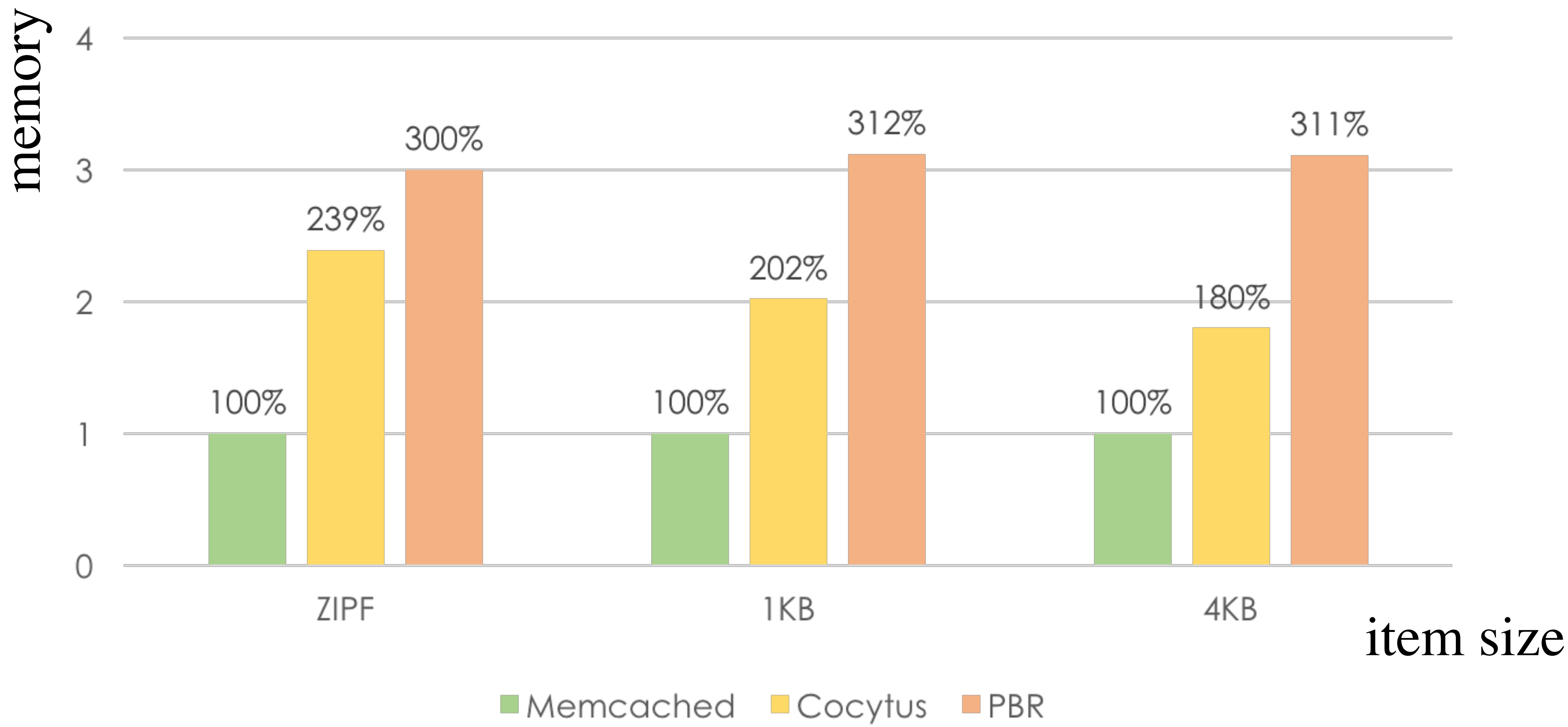
- 5 EC-Groups for Cocytus, each contains 3 DPs and 2 PPs
- 15 primary processes and 30 backup processes for primary-backup replication version
- 15 original processes for Memcached

1 node for client, 20 cores

- Run YCSB benchmark with 80 threads

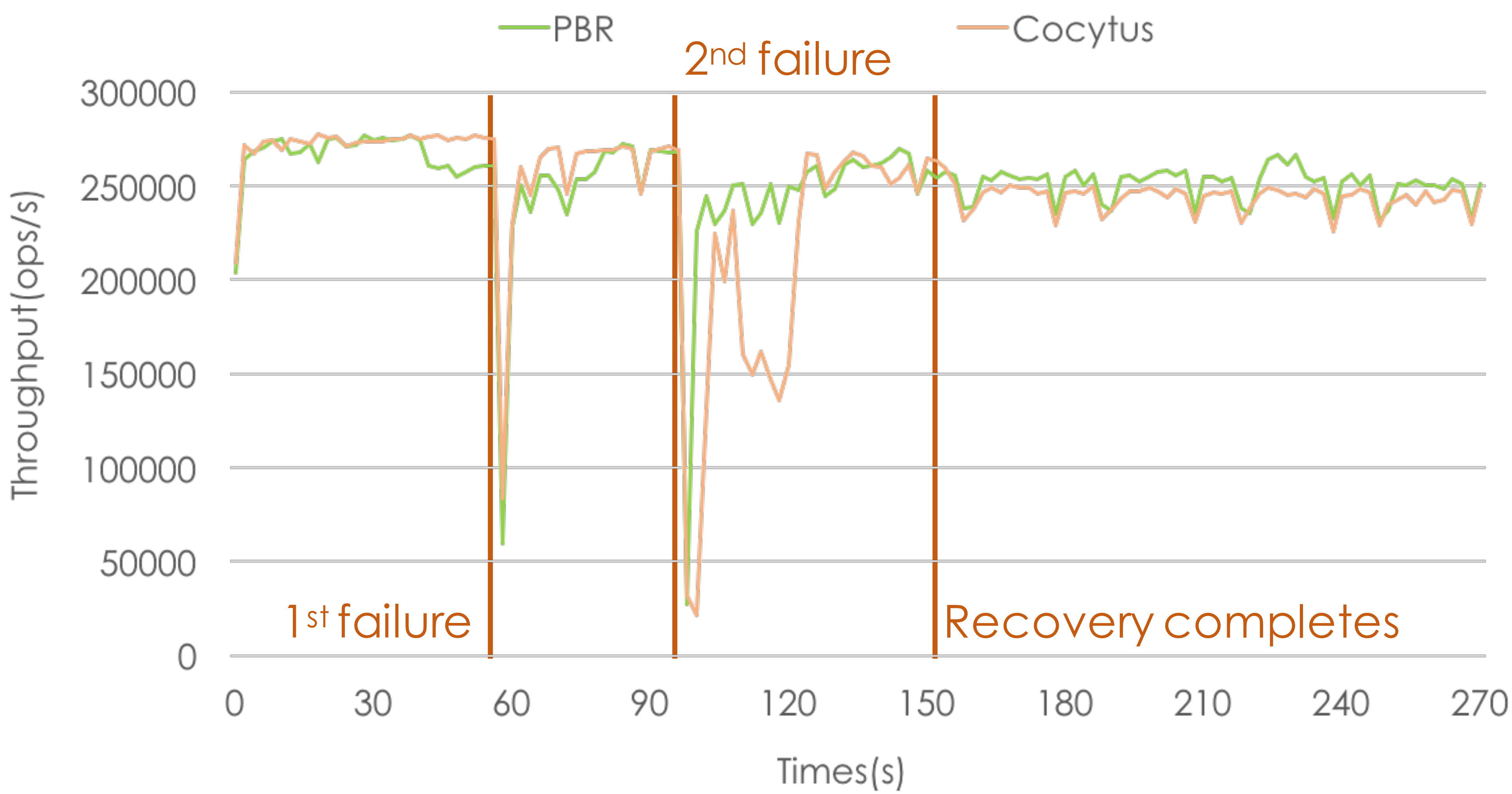
10Gbps network

Memory Consumption



**ZIPF: Zipfian distribution over the range from 10B to 1KB*

Recovery



(R:W=95%:5% & 1KB-size value & 12GB data/node)

CPU Overhead

Read:Write	Memcached	PB Replication		Cocytus	
	15 processes	15 primary processes	30 backup processes	15 data processes	10 parity processes
50%:50%	231%CPUs	**439%CPUs**	189%CPUs	**802%CPUs**	255%CPUs
95%:5%	228%CPUs	**234%CPUs**	60%CPUs	**256%CPUs**	54%CPUs
100%:0%	222%CPUs	**230%CPUs**	21%CPUs	**223%CPUs**	15%CPUs

Conclusion

- Replication approach is memory-consuming for in-memory KV-Stores
- Cocytus combines **erasure coding and replication** to achieve efficient and available in-memory KV-Store
- Cocytus could achieve better memory efficiency with low overhead compared with primary-backup replication on read-mostly workloads