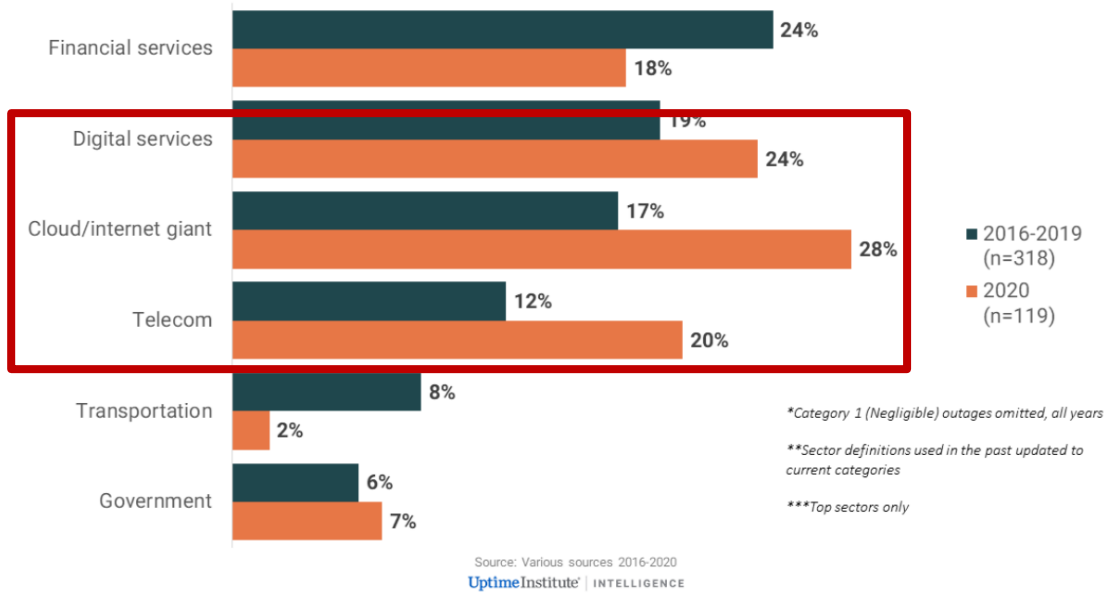

# Erasure Codes in Data Centers

## 数据中心高可靠纠删码技术

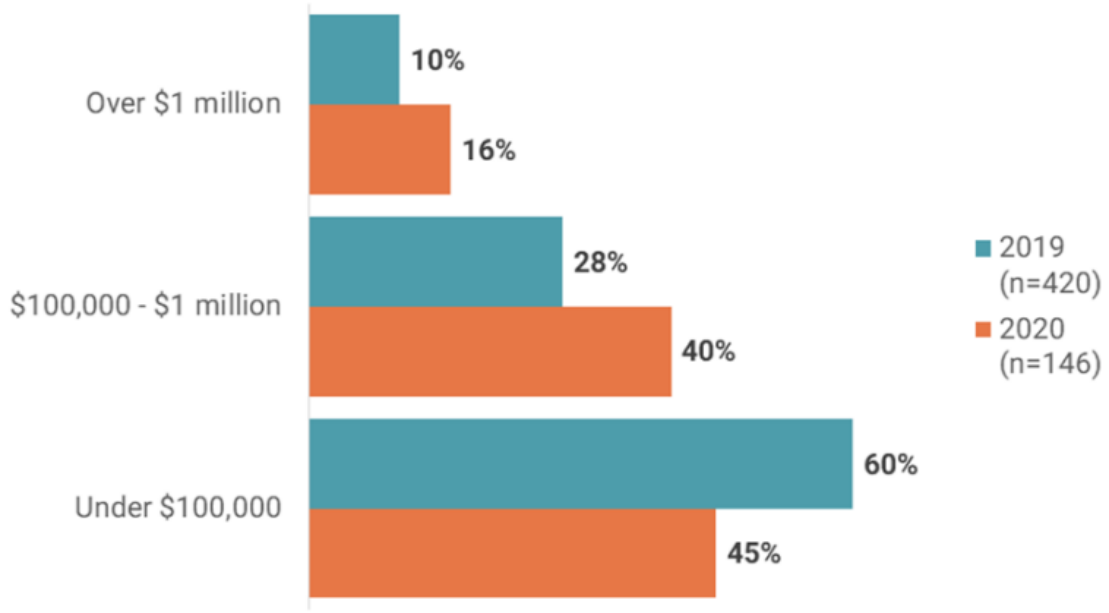
沈志荣 厦门大学

# Introduction

- Data volume is growing explosively
  - Failures arise unexpectedly yet prevalently
  - Fault tolerance is critical



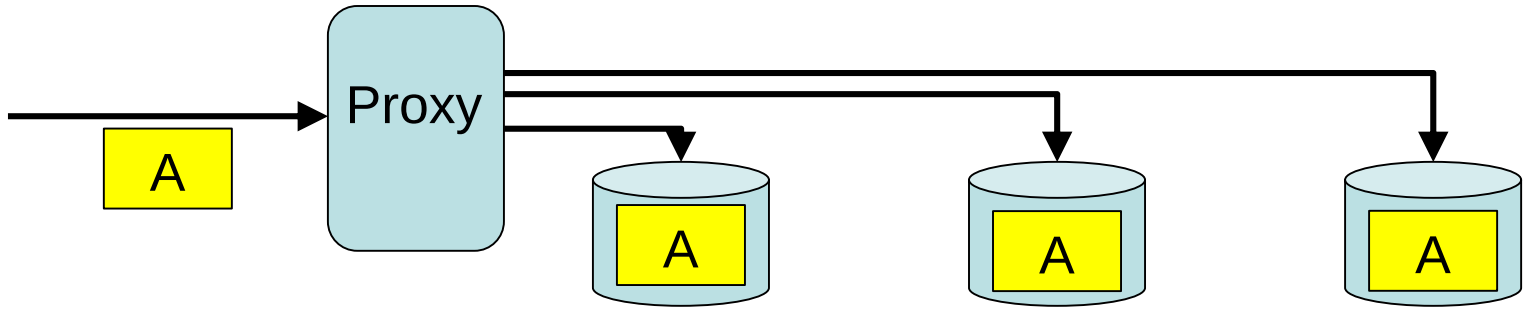

Commercial providers accounted for 72% of outages in 2020




Monetary loss caused by failures

# Two Redundancy Techniques

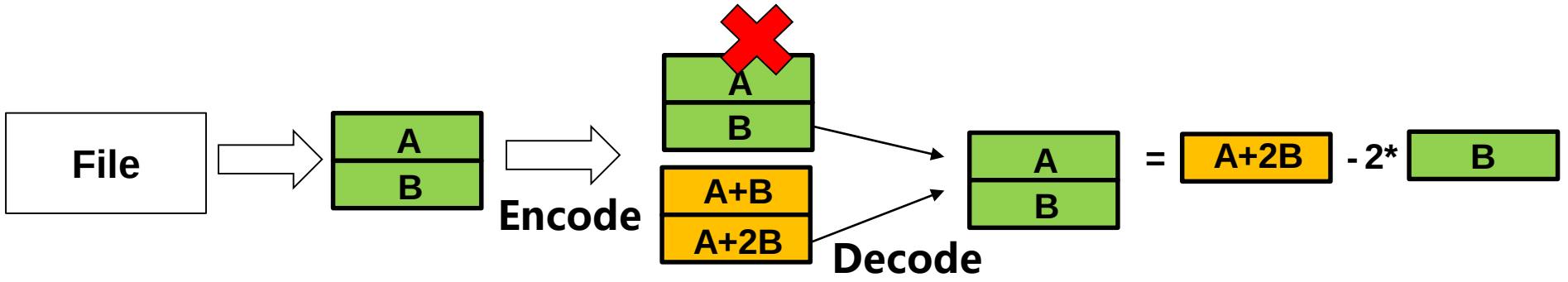
- Replication: directly keep multiple copies across different nodes
  - Triple replication requires 3x of storage redundancy
  - Tolerate n-1 failures in n-replication



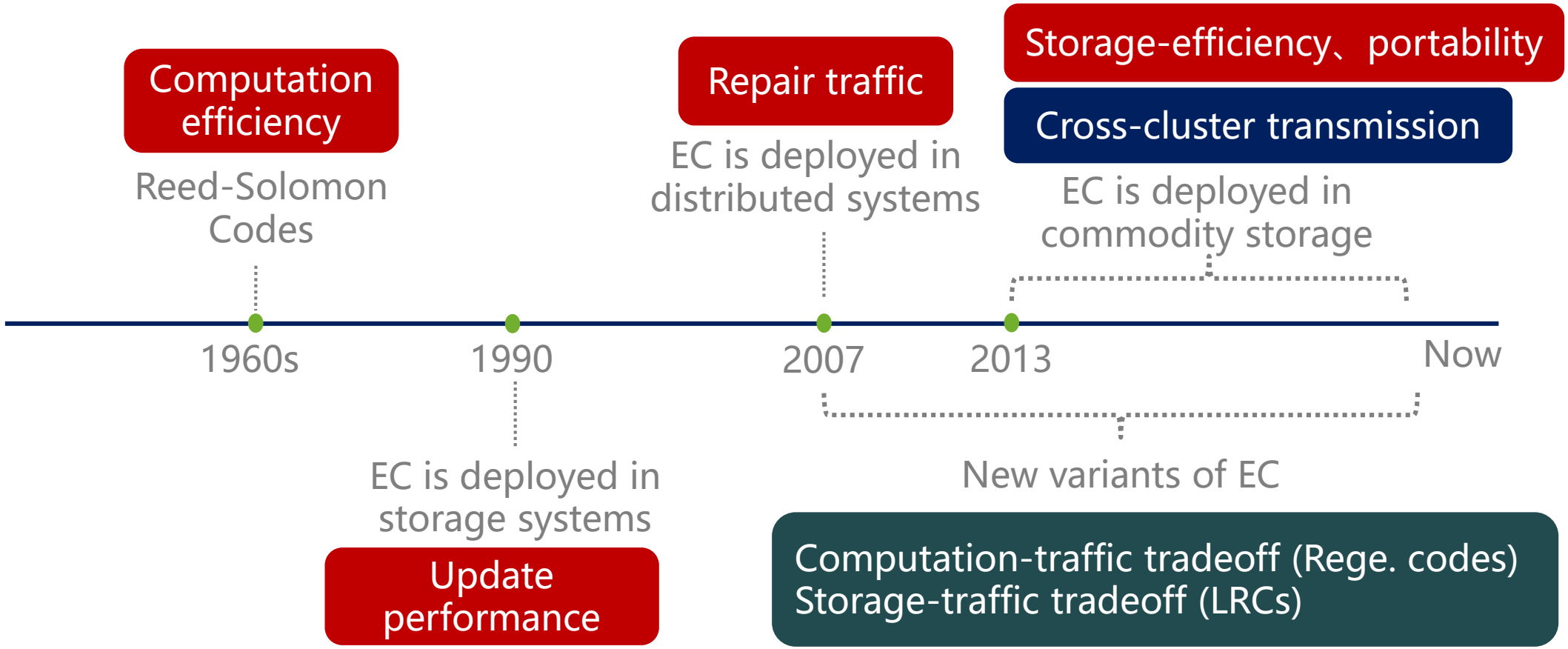

- **Erasure coding**: introduce slightly computational operations
  - Lower storage overhead with the same reliability guarantee as replication
  - Deployed in Google, Facebook, etc.

# Erasure Coding

- Divide a data file to **k** data chunks
- Encode k chunks to another redundant **m** parity chunks
- Distribute k+m chunks (forming a stripe) across k+m nodes
- Tolerate *any* m nodes failures

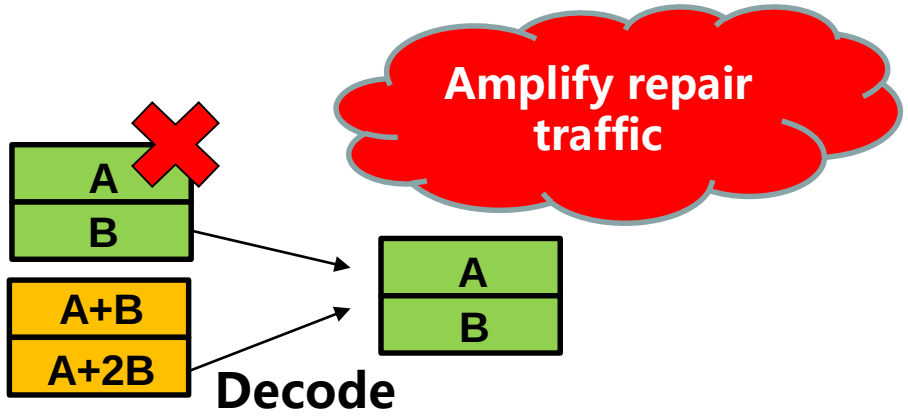


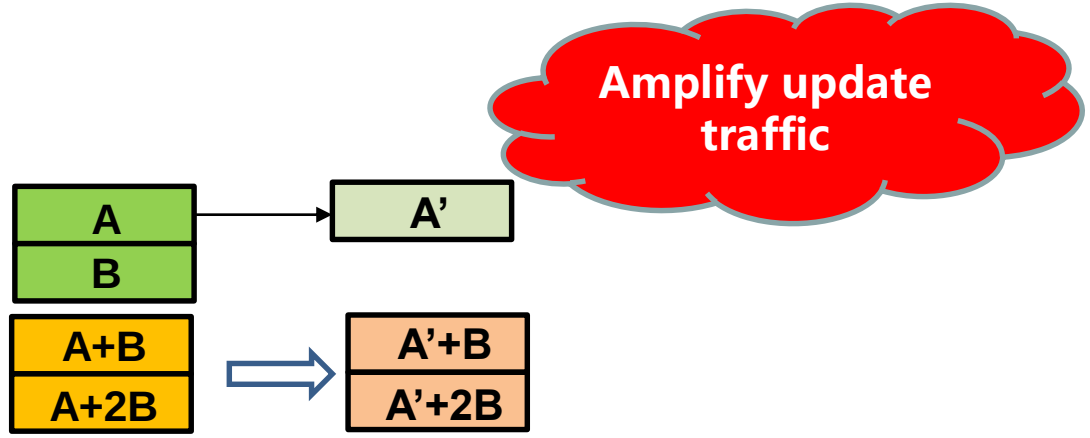
# Erasure Coding



# Shortcomings of EC

- Erasure coding reduces storage overhead at the expense of I/O amplification in both repair and update
  - Repairing a single chunk needs **k** surviving chunks
  - Updates a data chunk calls for the recalculation of **m** parity chunks



# Our Works

- In this talk, I will introduce our recent studies
  - Boosting Full-Node Repair in Erasure-Coded Storage [USENIX ATC'21]

    A repair-scheduling framework that boosts full-node repair for a variety of erasure codes and repair algorithms

  - Optimal Rack-Coordinated Updates in Erasure-Coded Data Centers [INFOCOM'21]

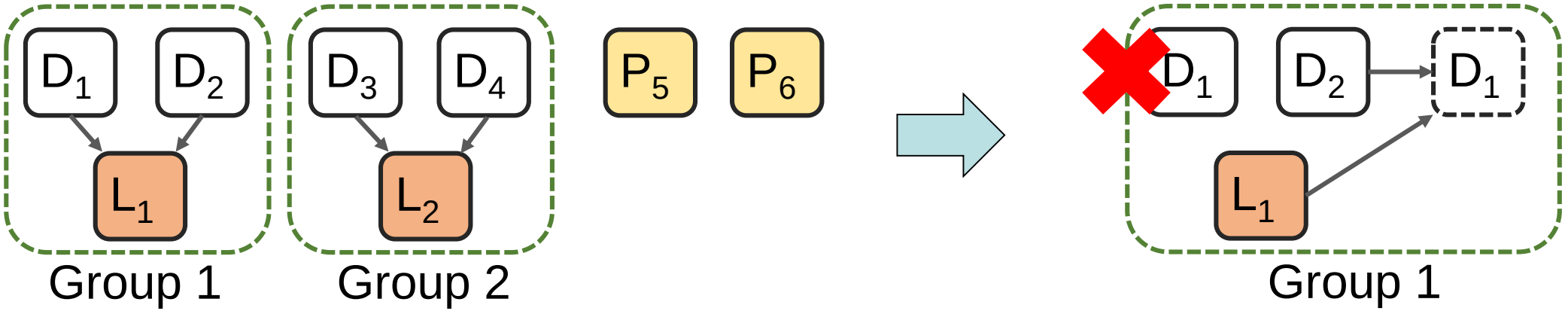
# Erasure Coding

- Drawback: substantial repair traffic
  - Retrieve k chunks to repair a single failed chunk

- Relieve the I/O amplification problem in repair
  - Repair-efficient codes with reduced repair traffic (What to retrieve?)
    - Locally Repairable Codes [ATC'12, PVLDB'13]
    - Regenerating Codes [TIT'10, TIT'11]
  - Efficient repair algorithms to parallelize the repair process (How to retrieve?)
    - Partial-Parallel-Repair (PPR) [EuroSys'16]
    - Repair pipelining (ECPipe) [ATC'17]

# Repair-Efficient Codes

- Locally Repairable Codes (LRCs)
  - Generate *local parity chunks* to facilitate repair at the expense of additional storage cost




(k, l, m) = (4, 2, 2)

Retrieve two chunks for repair

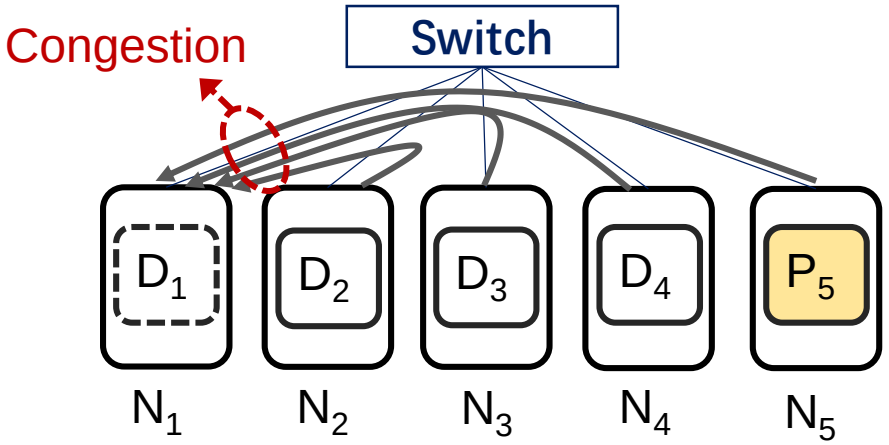
# Repair Algorithms

- Single-chunk repair algorithm
  - Accelerate the repair without reducing the repair traffic
  - Introduce transmission dependency

T1: $N_3 \longrightarrow N_2$, $N_5 \longrightarrow N_4$

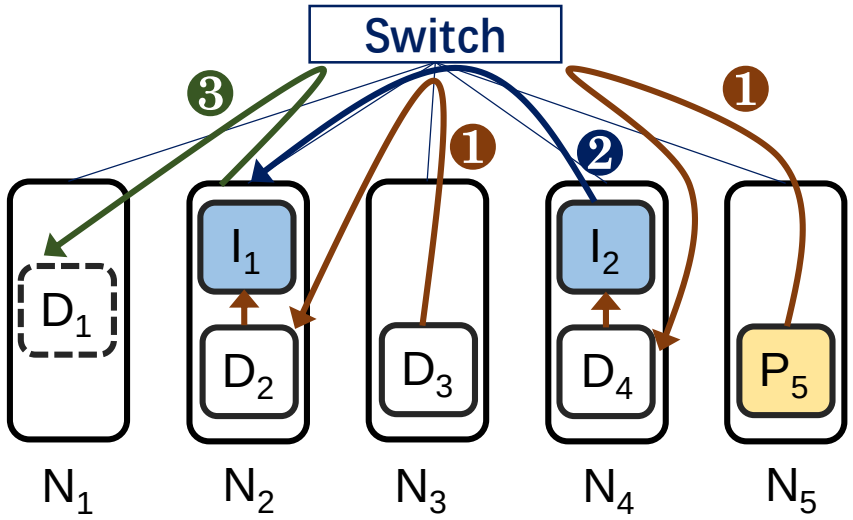
T2: $N_4 \longrightarrow N_2$

T3: $N_2 \longrightarrow N_1$




Conventional Repair (CR)

Repair time : 4 timeslots



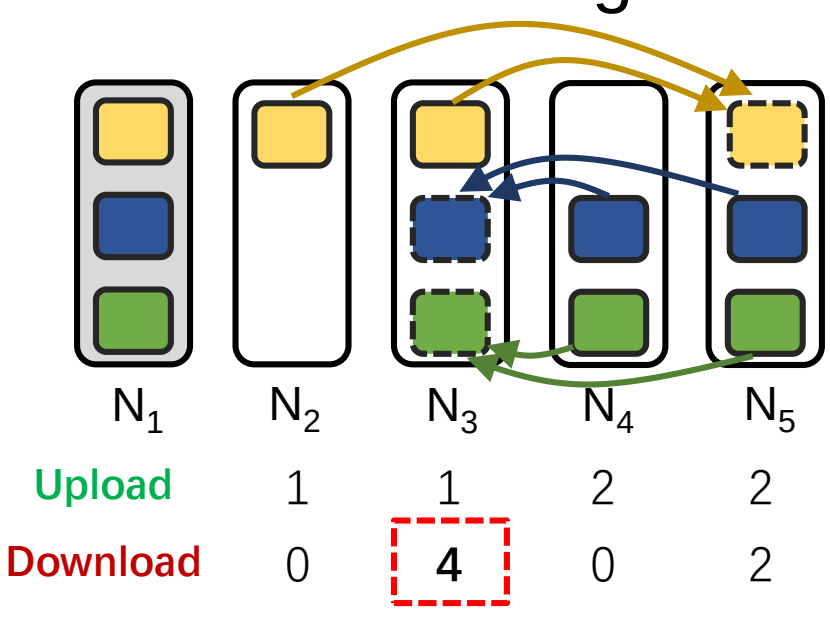

Partial-Parallel-Repair (PPR)

☺ Repair time : $log_2(4+1) = 3$ timeslots

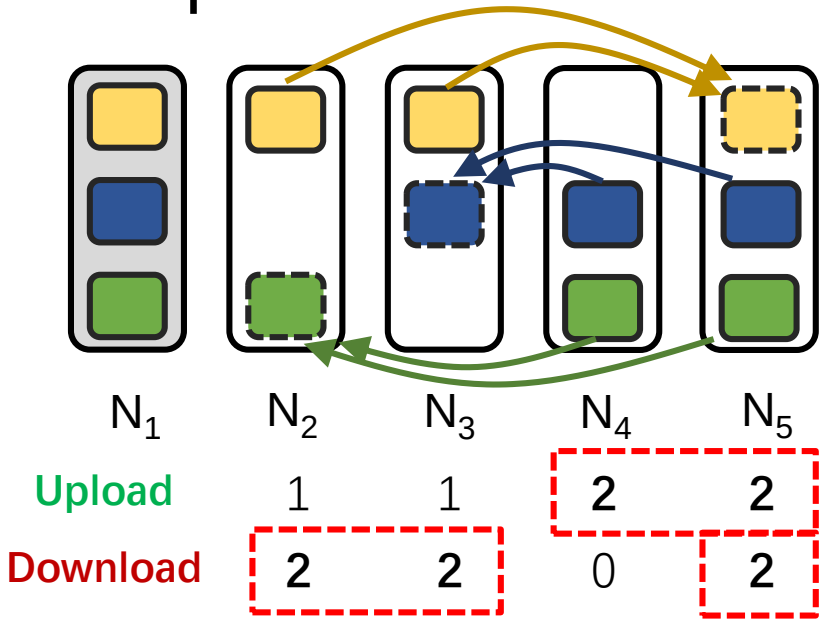
☹ Introduce **transmission dependency**: $D_4$ should wait for $P_5$ for aggregation

# Motivation

➢ **Limitation 1:** Failing to utilize the full duplex transmission




(a) Unbalanced repair solutions



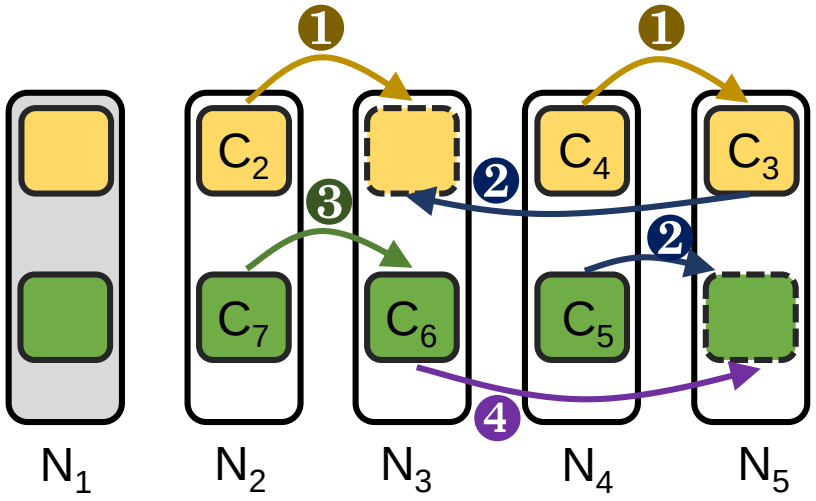

(b) Balanced repair solutions

Two chunks' repair under the conventional repair (CR)

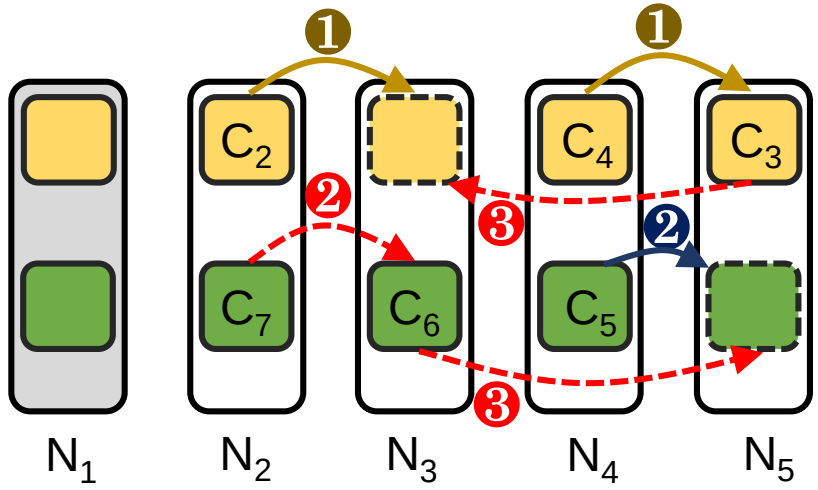
The repair time is determined by the most loaded node

# Motivation

➢ **Limitation 2:** Failing to fully utilize the bandwidth at each timeslot




(a) Repair using four timeslots




(b) Repair using three timeslots

Two chunks' repair under the partial-parallel-repair (PPR)

Transmission scheduling affects bandwidth utilization

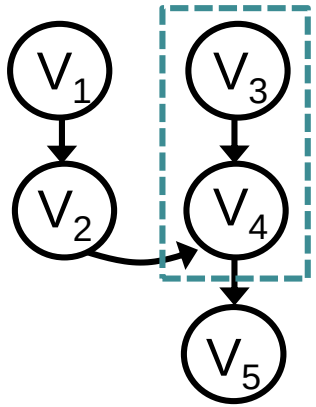
# Our Contributions

- **RepairBoost:** a framework to speed up the full-node repair
  - Tech#1: Repair abstraction (for generality and flexibility)
  - Tech#2: Repair traffic balancing (for load balancing)
  - Tech#3: Transmission scheduling (for saturating bandwidth utilization)
- A prototype RepairBoost integrated with HDFS
- Tackle multiple node failures and facilitate the repair in heterogeneous environments
- Experiments on Amazon EC2
  - Increase the repair throughput by 35.0-97.1%

# Repair Abstraction

- Formalize a single-chunk repair through a *repair directed acyclic graph* (RDAG)
  - Characterize the data routing over the network and the dependencies among the requested chunks
  - e.g., for RS(k, m),  k+1 vertices
    - $\{v_1, v_2, \cdots, v_k\}$: k nodes that retrieve chunks
    - $v_{k+1}$ : destination node for repairing the lost chunk
  - Directed edges represent the data routing directions specified in repair algorithms

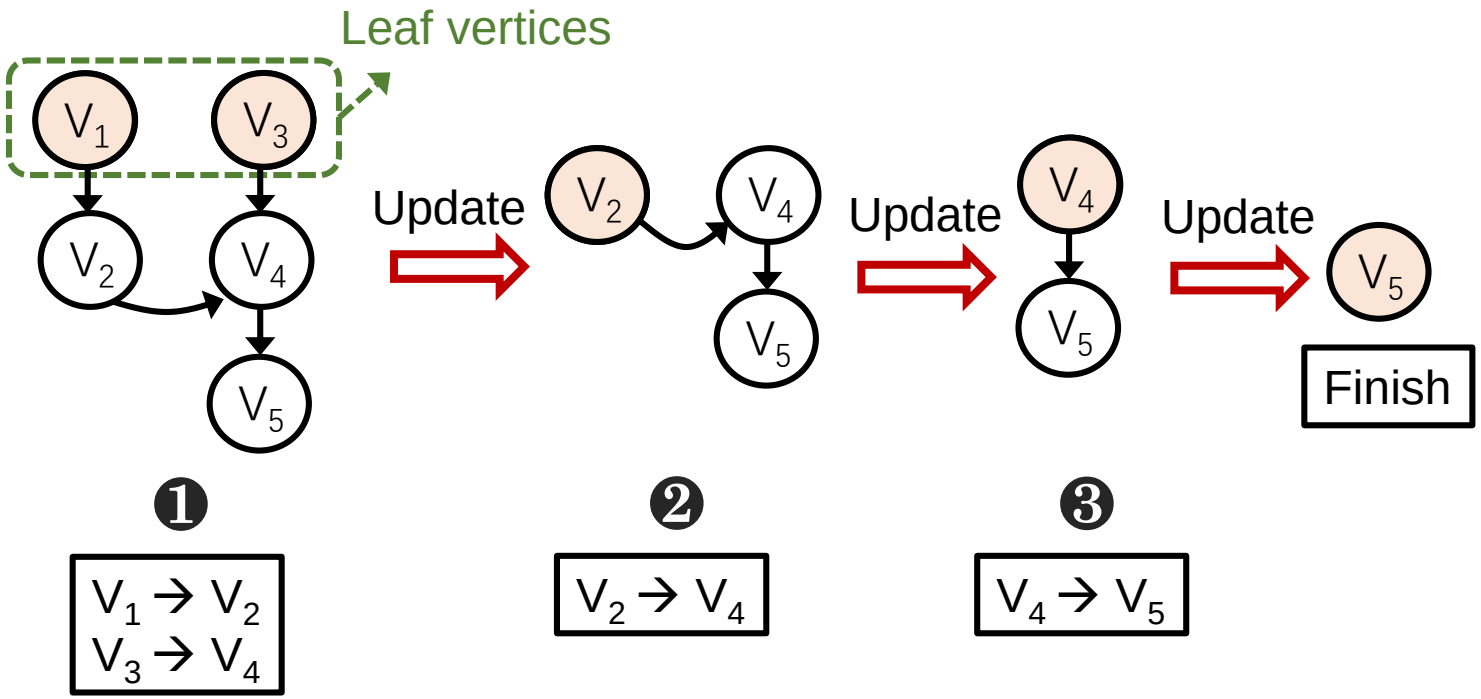
**An RDAG of PPR when k=4**




① $V_3$ is a child of $V_4$
② $V_4$ should collect all its children before sending its data to its parent (i.e., $V_5$)

# Repair Abstraction

- Repair process guided by RDAG
  - The repair starts from the *leaf vertices* (without predecessor dependency)
  - As the repair proceeds, iteratively remove edges and vertices from an RDAG



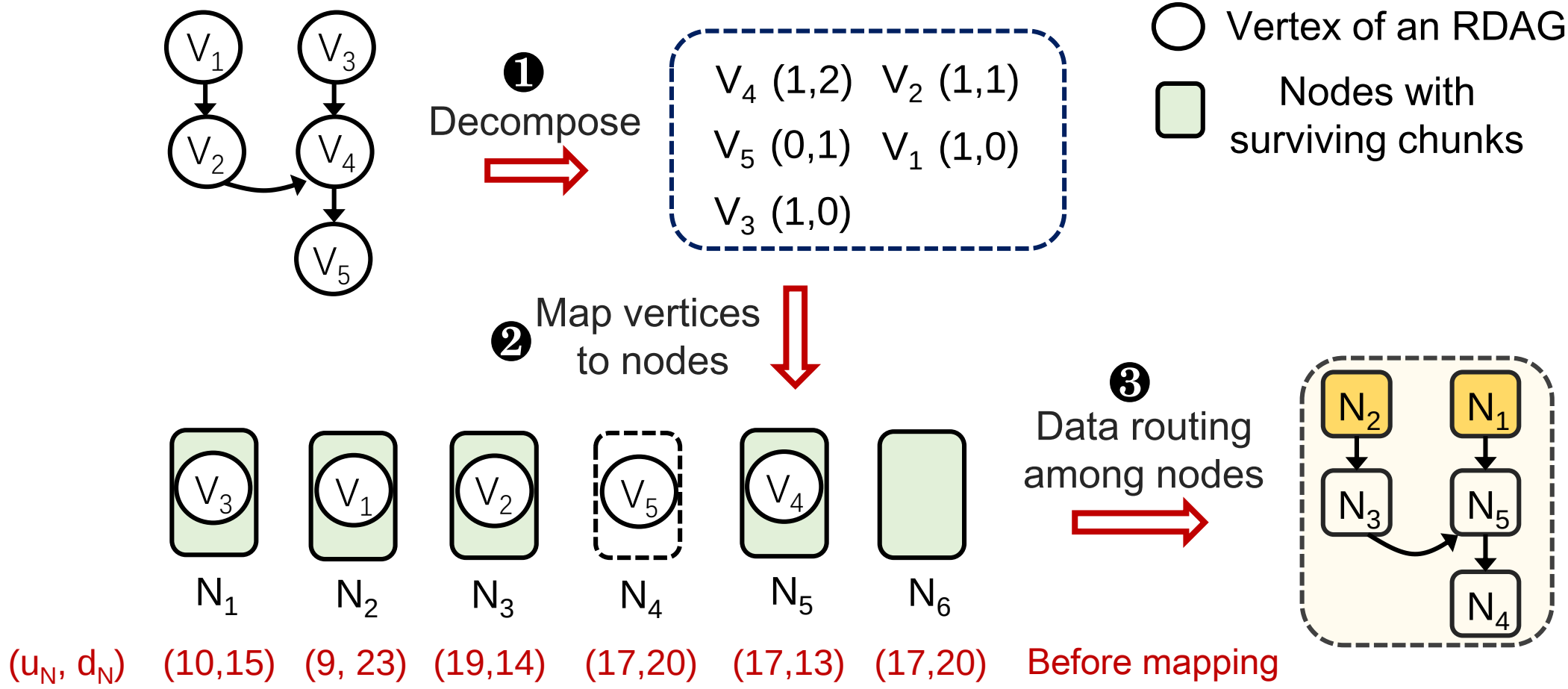
# Repair Traffic Balancing

- Decompose RDAGs into vertices (with different upload and download traffics) and map the vertices to storage nodes
  - Ob#1: Retaining fault tolerance degree
  - Ob#2: Balance the upload and download repair traffic

- The vertices of RDAGs are classified and given different priorities according to degree
  - Intermediate vertices ($u = 1 \text{ and } d > 0$)
  - Root vertex ($u = 0 \text{ and } d > 0$)
  - Leaf vertices ($u > 0 \text{ and } d = 0$)

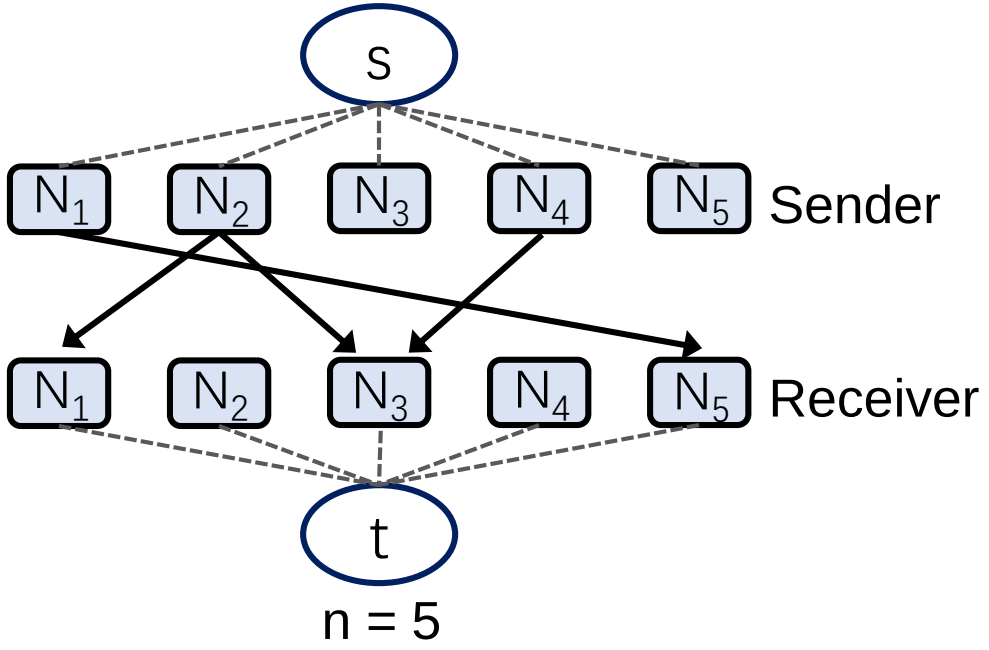
# Repair Traffic Balancing

- Example of mapping vertices of an RDAG to nodes



# Transmission Scheduling

- The bandwidth may not be utilized at each timeslot during repair (Limitation 2)
- Formulate as a maxflow problem
  - 2n+2 vertices
    - n senders: potentially send data for repair
    - n receivers: potentially receive data at the same time
  - Establish the connection between senders and receivers according to the RDAGs



# Transmission Scheduling

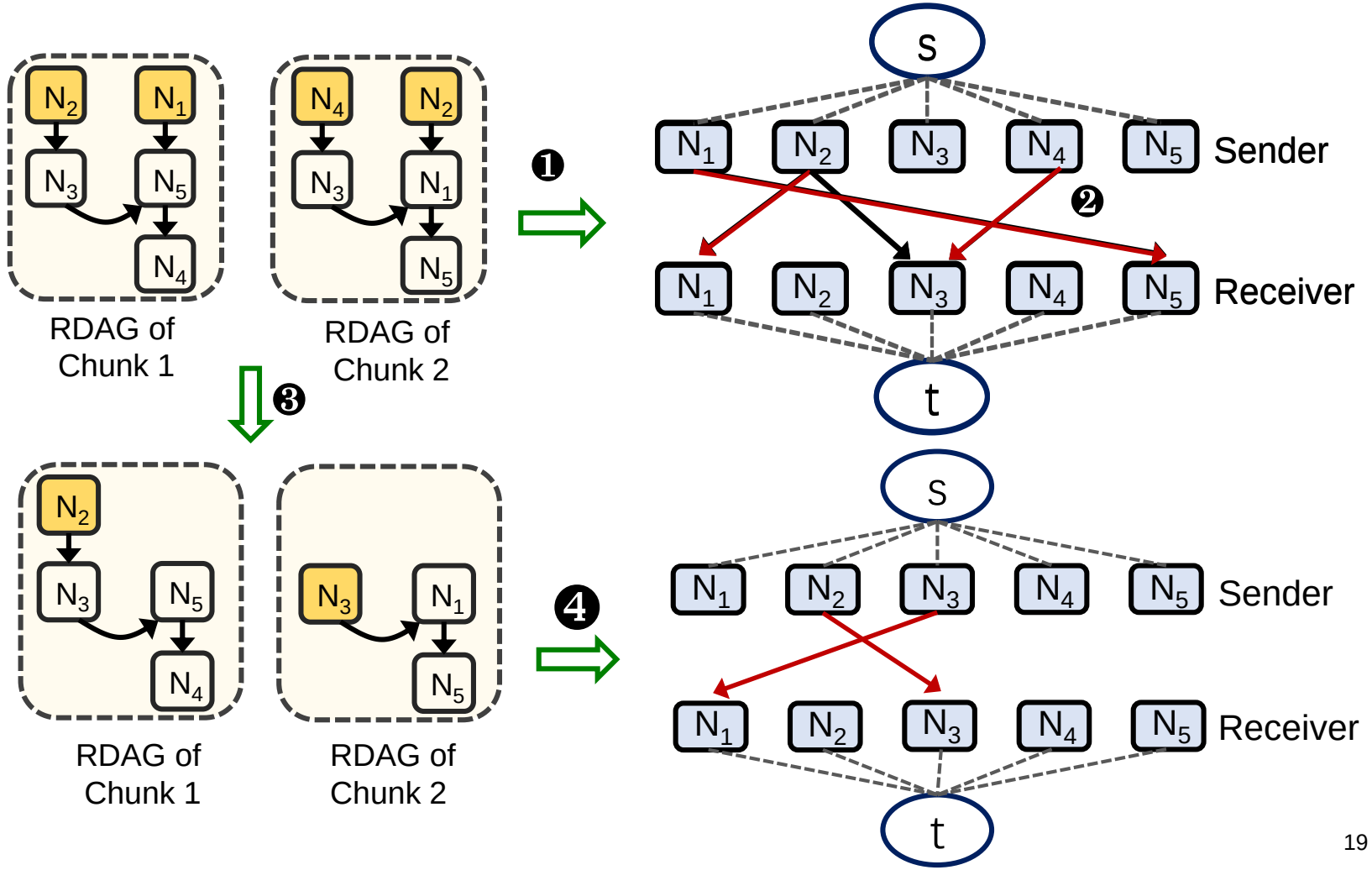
➢ Example of repairing two chunks among five surviving nodes

❶ Construct a network

❷ Establish a maximum flow

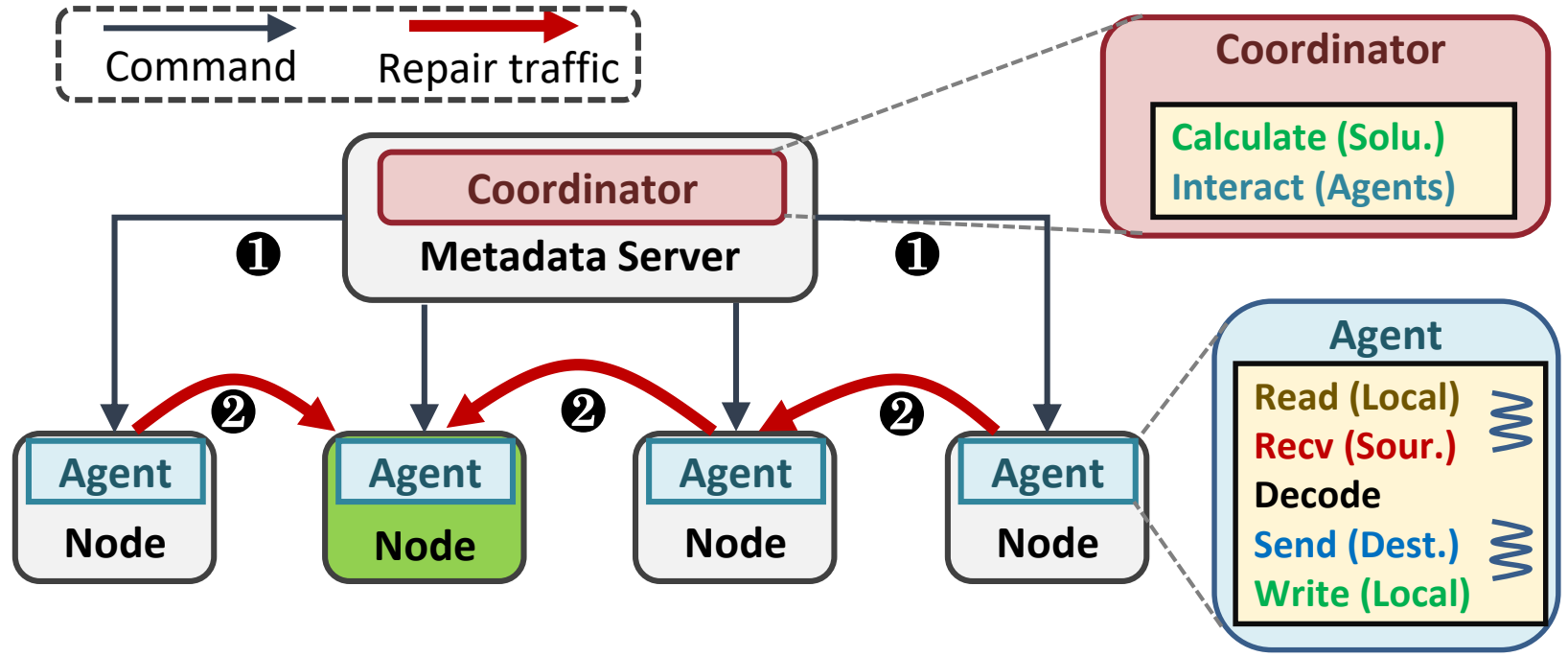
❸ Update the RDAG

❹ Construct a new network



# Implementation

- RepairBoost serves as an independent middleware running atop existing storage

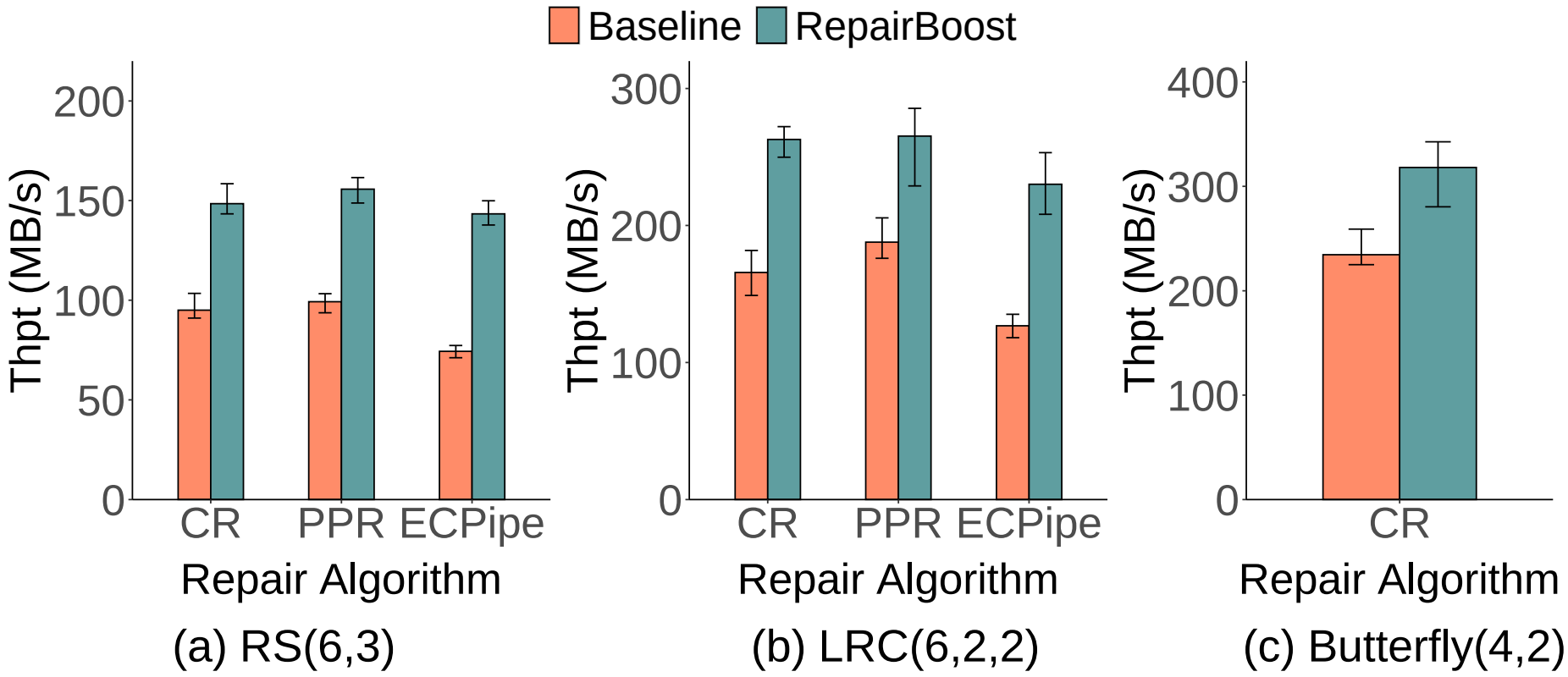



- The coordinator manages the metadata of stripes
- The agents are standby to wait for the repair commands and perform the repair operations cooperatively

# Evaluation Setup

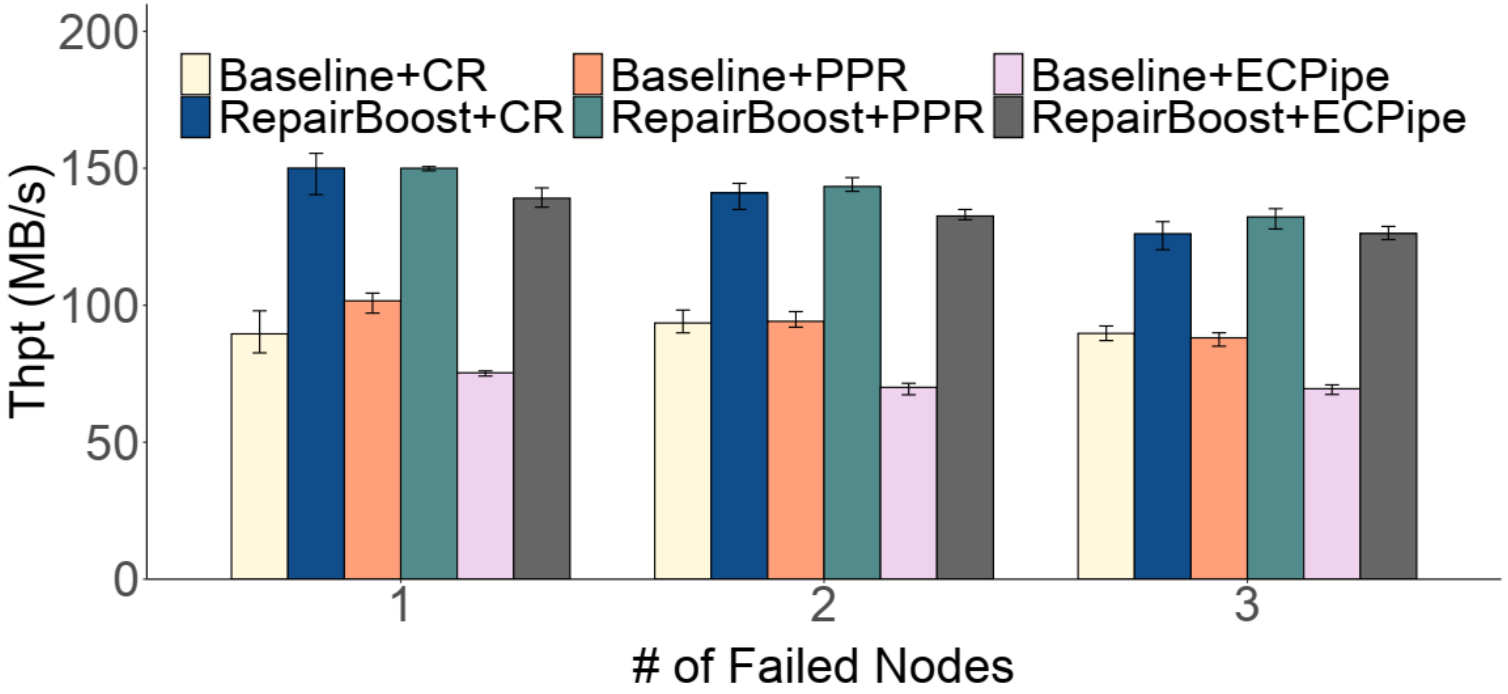
- Amazon EC2
  - 17 m5.large machines (1 coordinator and 16 agents)
- Default configurations
  - Chunk size: 64MB, Packet size: 1MB
  - RS(6, 3)
- Single-chunk repair algorithms
  - Conventional repair (CR)
  - Partial-Parallel-Repair (PPR)
  - Repair pipelining (ECPipe)
- Baseline: random selection
- Metric: repair throughput (size of data repaired per time unit)

# Performance Results




- Ob#1: Butterfly(4,2) reaches the highest repair throughput
  - as it needs to fetch only half of the data
- Ob#2: RepairBoost can improve the repair throughput by an average of 60.4% for different erasure codes

# Multi-Node Repair




- Ob#1: RepairBoost improves the repair throughput by 39.5% (a single node failure) and by 35.7% (triple node failures)
- Ob#2: The repair throughput of RepairBoost drops slightly when more nodes fail
  - Fewer selected nodes can participate in the repair

# Conclusion

- RepairBoost, a scheduling framework that boosts the full-node repair for various erasure codes and repair algorithms
  - Employ graph abstraction for single-chunk repair
  - Balance the upload and download repair traffic
  - Schedule the transmission of chunks to saturate unoccupied bandwidths

- Source code:

  https://github.com/shenzr/repairboost-code

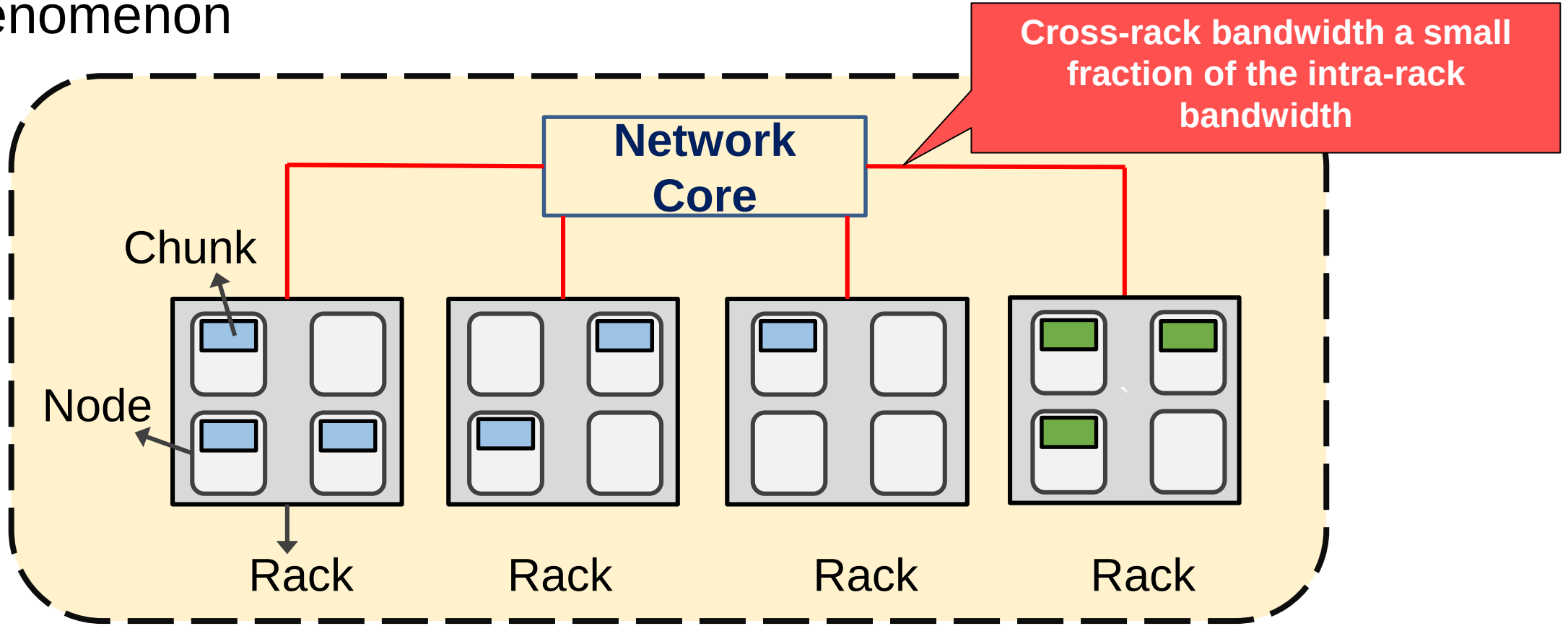
# Our Works

- In this talk, I will introduce our recent studies
  - Boosting Full-Node Repair in Erasure-Coded Storage [USENIX ATC'21]
  - Optimal Rack-Coordinated Updates in Erasure-Coded Data Centers [INFOCOM'21]

An update approach that minimizes the cross-rack update traffic in erasure-coded data centers

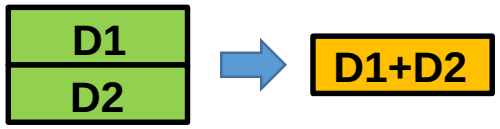
# Data Center

- Hierarchical architecture results in the bandwidth diversity phenomenon



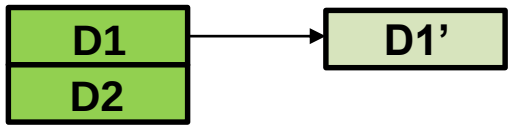
# Parity Update

- Each parity chunk $P_j$ can be calculated as:




$$P_j = \sum_{i=1}^{k} \gamma_{i,j} D_i$$

- When data chunk $D_h$ is update to $D'_h$ each parity chunk could be update as:




$$P_j^{'} = P_j + \gamma_{h,j}(D_h^{'} - D_h) = P_j + \gamma_{h,j}\Delta D_h$$

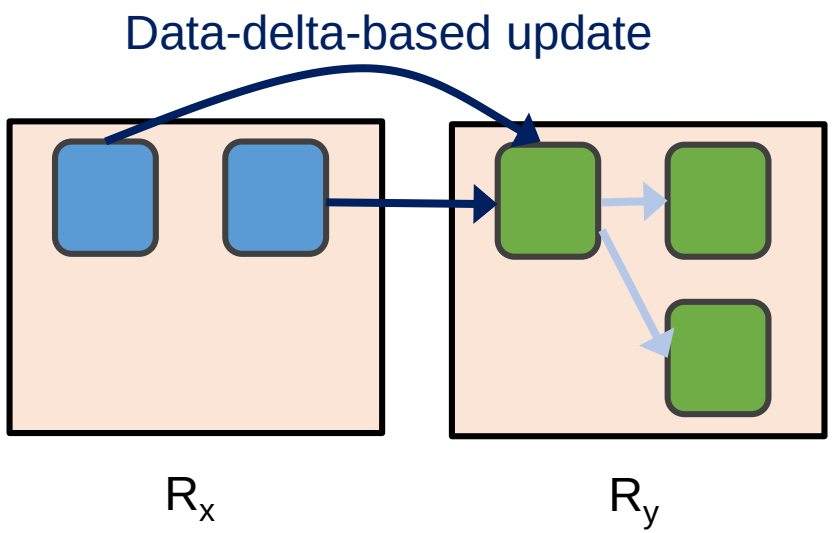
data delta chunk

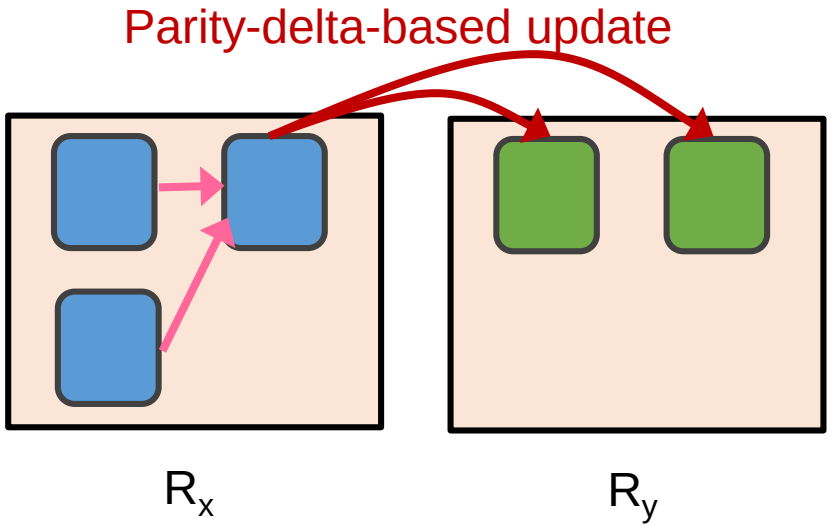
parity delta chunk

# Parity Update

- Selective parity update: data-delta-based update and parity-delta-based update

$$P_j^{'} = P_j + \gamma_{h,j}\Delta D_h = P_j + \Delta P_j$$




When the updated data chunks in $R_x$ is fewer than the parity chunks in $R_y$

When the updated data chunks in $R_x$ is no fewer than the parity chunks in $R_y$

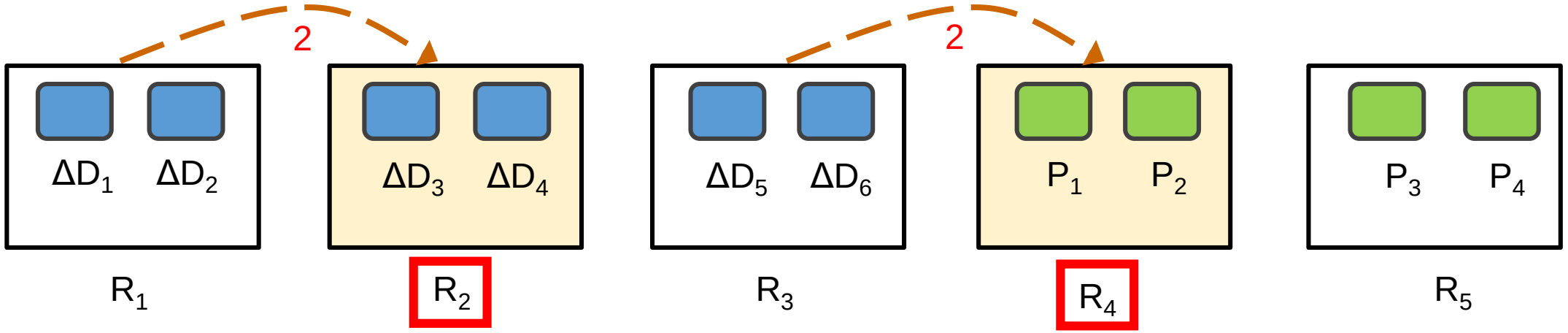
# Our Contribution

- RackCU: an optimal **Rack**-**C**oordinated **U**pdate solution that reaches the lower bound of the cross-rack update traffic
  - Breaks the whole parity update procedure into a delta-collecting phase and another selective parity update phase
  - Reliability: allow racks to update parity chunks immediately after data chunks are updated

- Large-scale simulation, and Alibaba Cloud ECS experiments
  - Show that RackCU reduces 22.1%-75.1% of cross-rack update traffic and hence increases 34.2%-292.6% of update throughput

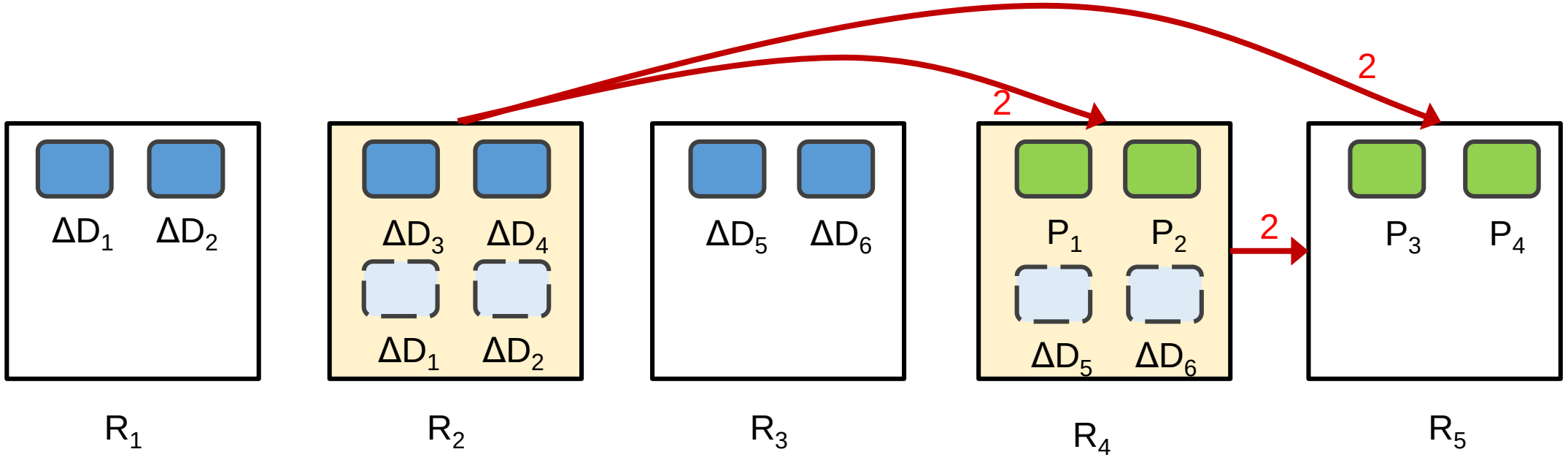
# RackCU Design

- Delta-collecting phase: select collector racks and each collector is responsible for retrieving data delta chunks from the specified data racks



# RackCU Design

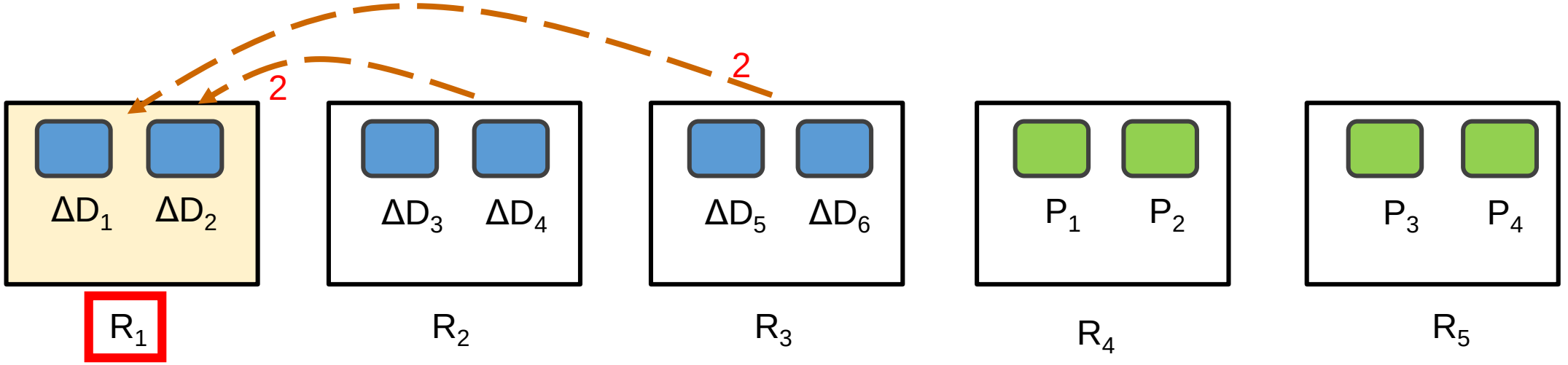
- Selective parity update phase: collector rack choose either the data-delta-based update or the parity-delta-based update to renew the parity chunks




Needs **10 chunks** transmitted across racks

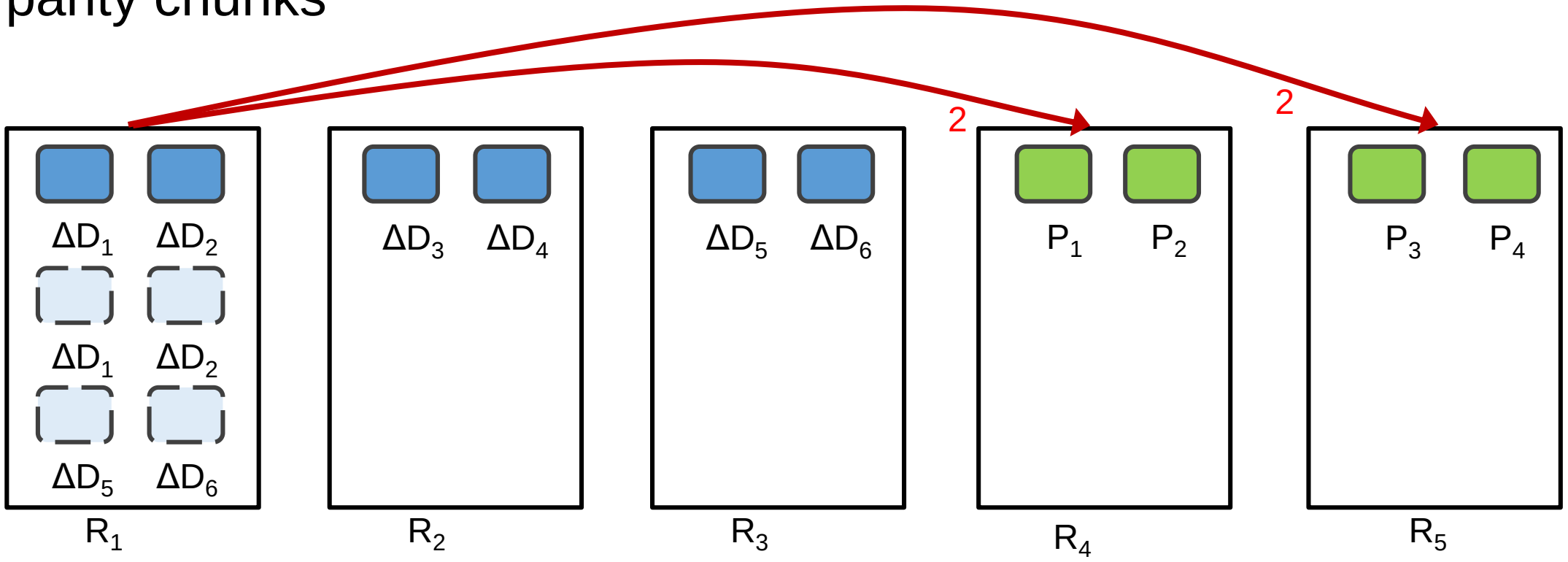
# RackCU Design

- Optimal rack-coordinated update: select the rack with the most chunks as collector rack to minimize cross-rack traffic
- We provide a theoretical proof to demonstrate RackCU minimizes the cross-rack update traffic



# RackCU Design

- Selective parity update phase: the collector rack $R_1$ renew the parity chunks




Needs **8 chunks** transmitted across racks

# Evaluation

- Large-Scale Simulation

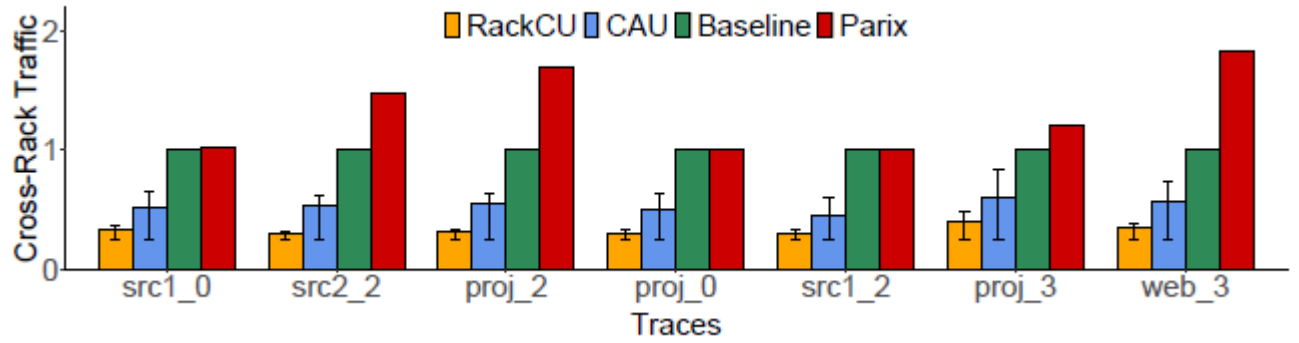
| Node | Rack | Erasure code | Chunk size |
|---|---|---|---|
| 200 | 10 | RS (12, 4) | 4KB |

- Testbed Experiments

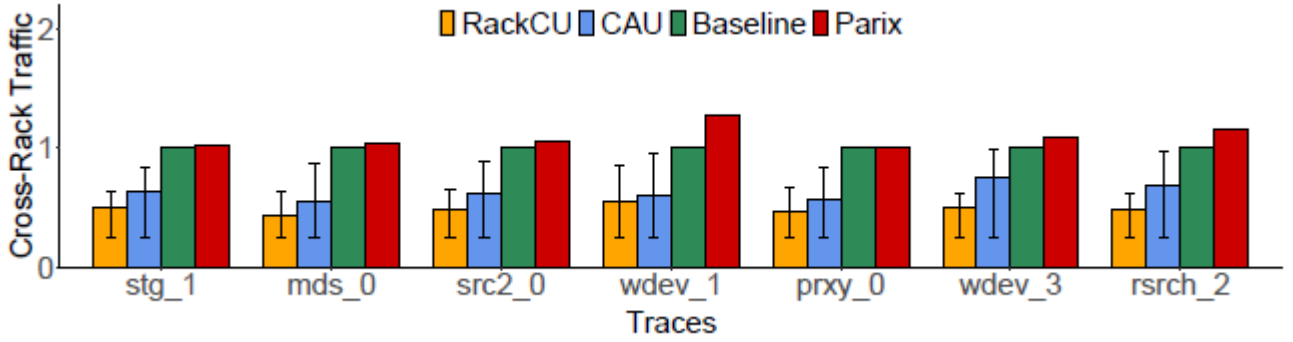
| Machine | CPU | Memory | OS | Bandwidth |
|---|---|---|---|---|
| 18 virtual machine instances (ecs.g6.large) | 2 vCPU (2.5GHz Intel Xeon Platinum 8269CY) | 8GB | Ubuntu 18.04 | 3 Gb/s |

| Node | Rack | Erasure code | Chunk size |
|---|---|---|---|
| 16 | 8 | RS (12, 4) | 4KB |

# Impact of update size



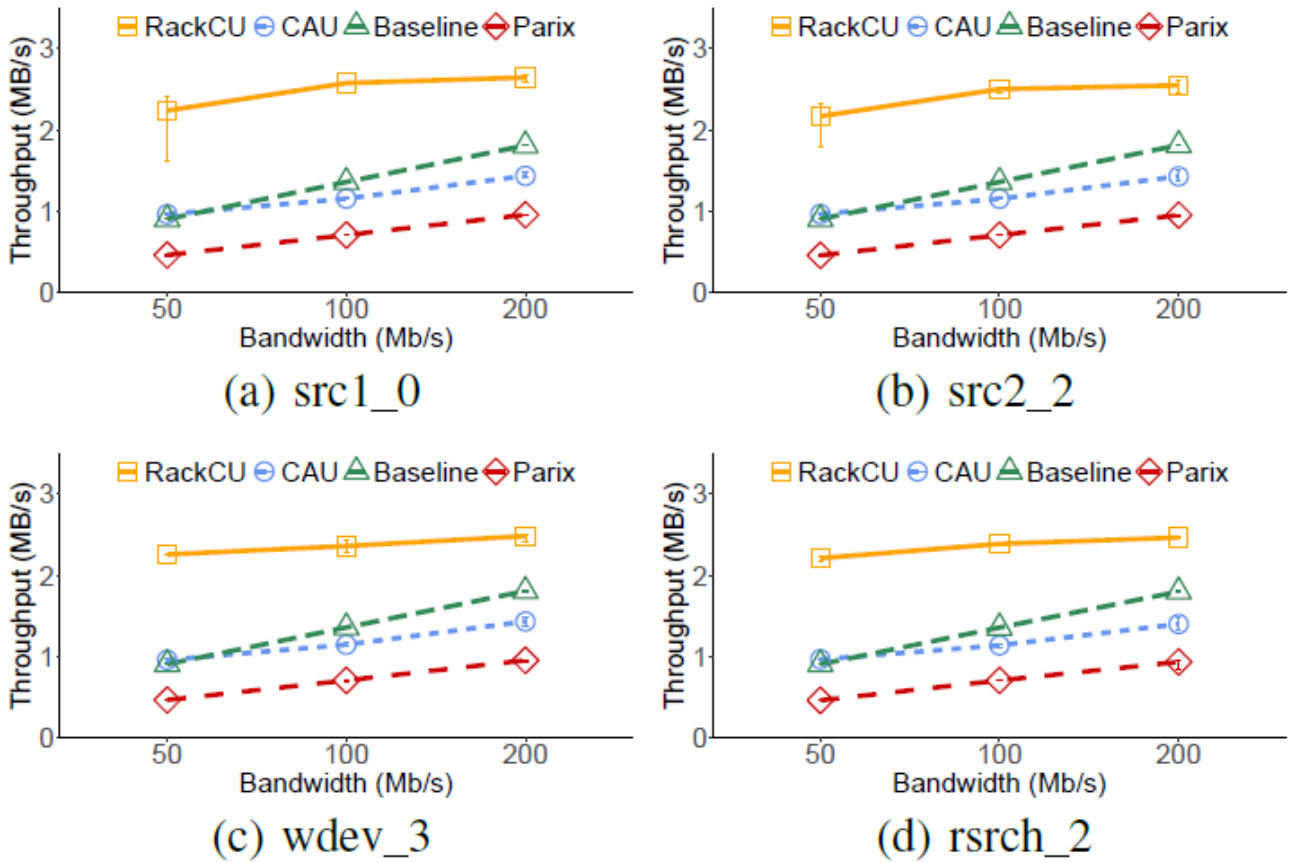

**Larger update sizes**




**Smaller update sizes**

- **Ob1:** RackCU can save the most cross-rack traffic
- **Ob2:** RackCU is more advantageous with larger update sizes

# Impact of cross-rack bandwidth




(a) src1_0 (b) src2_2

(c) wdev_3 (d) rsrch_2

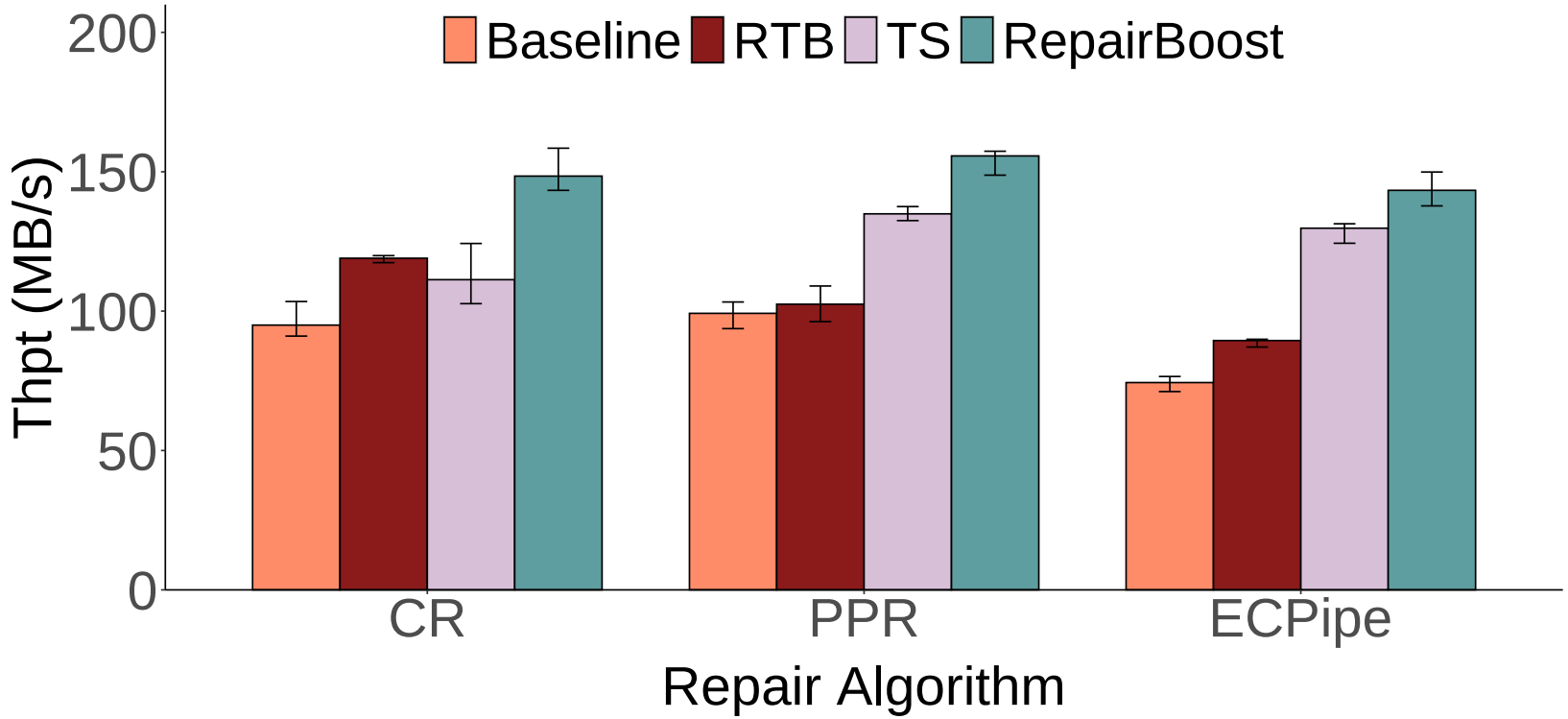
- **Ob:** RackCU improves the update throughput by 106.8%, 88.2%, and 262.2% when compared to CAU, the baseline, and Parix, respectively

# Conclusions

- **RackCU**: an optimal **Rack**-**C**oordinated **U**pdate solution
  - A delta-collecting phase and selective parity update phase
  - Reaches the lower bound of the cross-rack update traffic
- Simulation, and experiments
  - Large scale simulation
  - Testbed experiments on Alibaba Cloud ECS
- Source code of RackCU prototype:
  - **https://github.com/ggw5/RackCU-code**

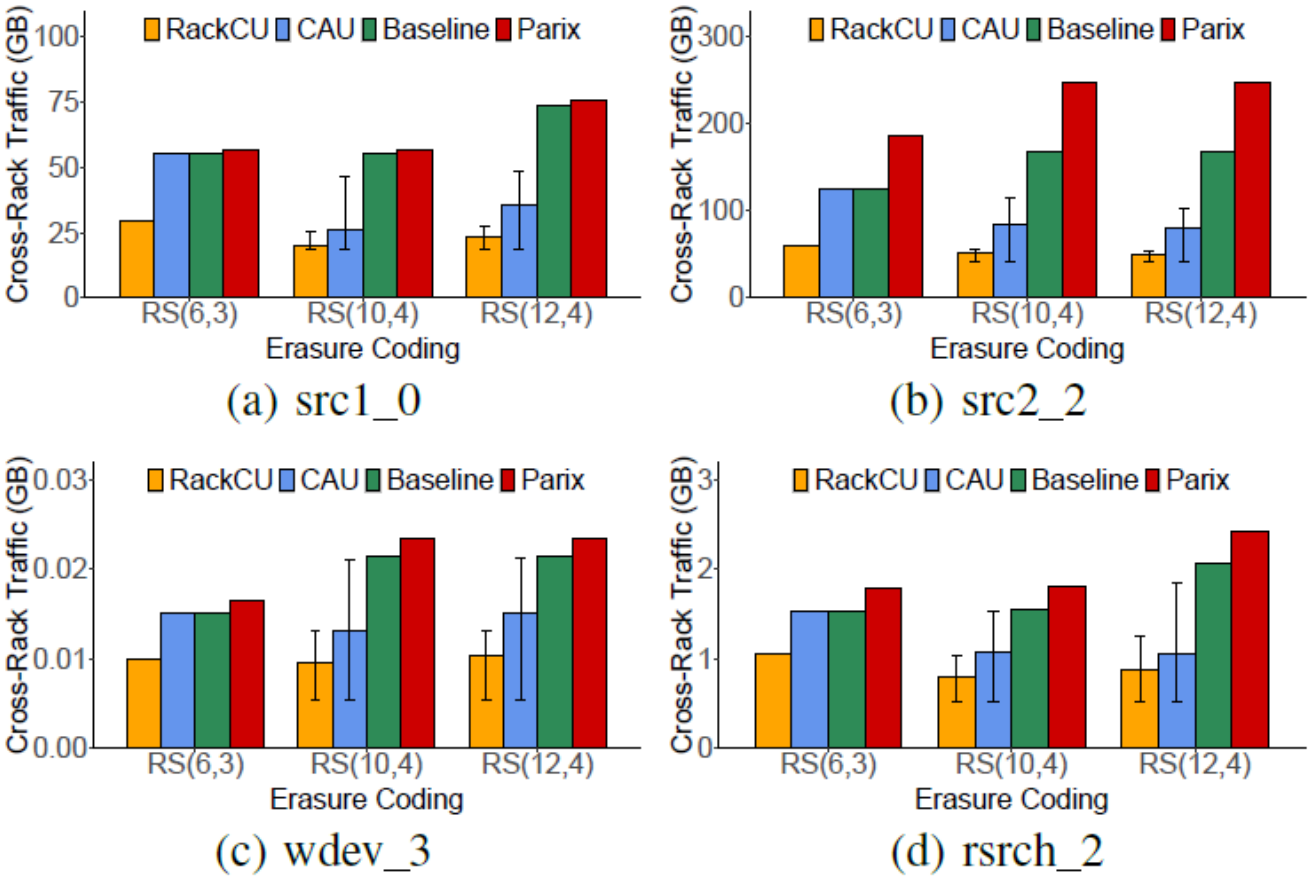
# Thank You!
# Q & A

# Breakdown Analysis




- Ob#1: The effectiveness of RTB and TS varies across different repair algorithms.
- Ob#2: RepairBoost achieves 45.7% and 19.8% higher repair throughput than RTB and TS, respectively.

# Impact of erasure coding




- **Ob:** RackCU reduce 33.3%, 54.1%, and 60.4% of the cross-rack update traffic on average compared to CAU, the baseline, and Parix, respectively