Repair Pipelining for Erasure-Coded Storage

Runhui Li, Xiaolu Li, **Patrick P. C. Lee**, Qun Huang

The Chinese University of Hong Kong

USENIX ATC 2017

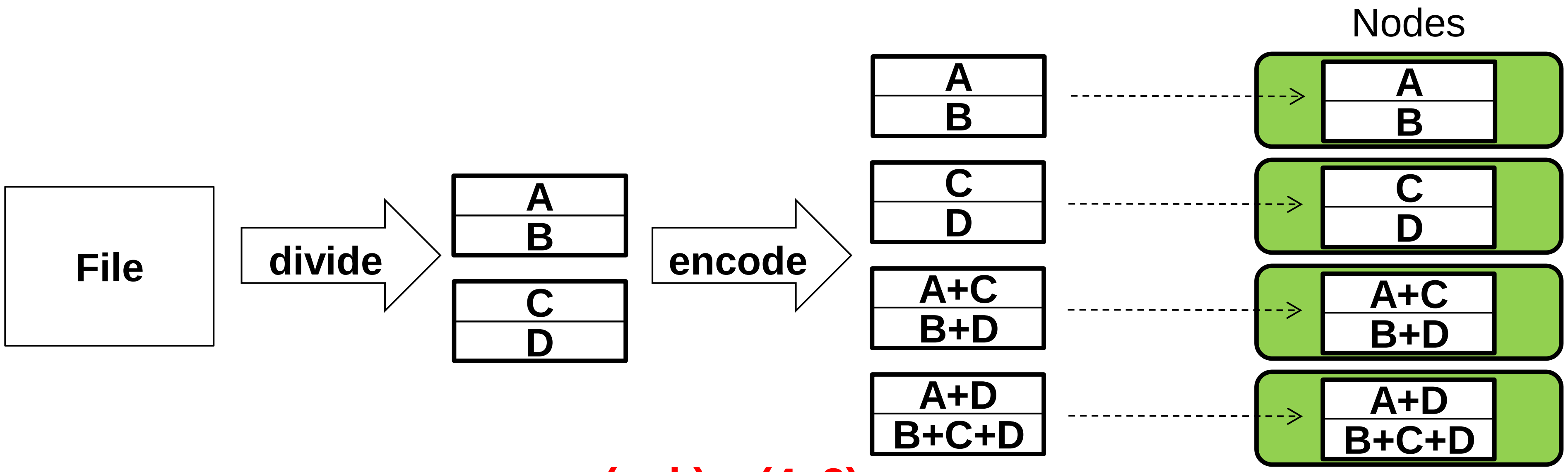
Introduction

- Fault tolerance for distributed storage is critical
  - **Availability**: data remains accessible under failures
  - **Durability**: no data loss even under failures

- **Erasure coding** is a promising redundancy technique
  - Minimum data redundancy via "data encoding"
  - Higher reliability with same storage redundancy than replication
  - Reportedly deployed in Google, Azure, Facebook
    - e.g., Azure reduces redundancy from 3x (replication) to 1.33x (erasure coding) → PBs saving

Erasure Coding

- Divide file data to **k** blocks
- Encode k (uncoded) blocks to **n** coded blocks
- Distribute the set of n coded blocks (stripe) to n nodes
- **Fault-tolerance**: any k out of n blocks can recover file data



(n, k) = (4, 2)

Remark: for systematic codes, k of n coded blocks are the original k uncoded blocks

Erasure Coding

- Practical erasure codes satisfy **linearity** and **addition associativity**
  - Each block can be expressed as a linear combination of any k blocks in the same stripe, based on Galois Field arithmetic
  - e.g., block $B = a_1B_1 + a_2B_2 + a_3B_3 + a_4B_4$
    for $k = 4$, coefficients a_i's, and blocks B_i's
- Also applicable to XOR-based erasure codes
- Examples: Reed-Solomon codes, regenerating codes, LRC

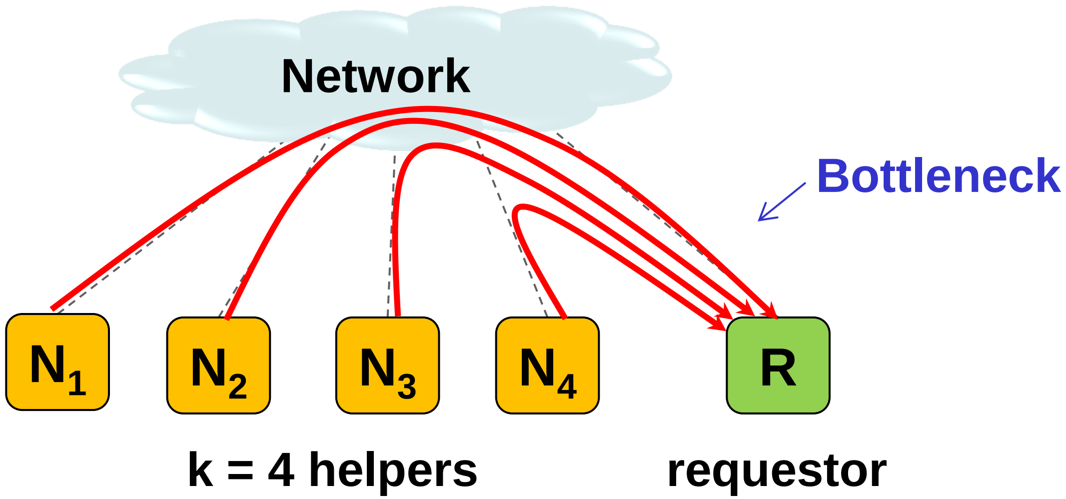
Erasure Coding

- **Good**: Low redundancy with high fault tolerance
- **Bad**: **High repair penalty**
  - In general, k blocks retrieved to repair a failed block

- Mitigating repair penalty of erasure coding is a hot topic
  - New erasure codes to reduce repair bandwidth or I/O
    - e.g., Regenerating codes, LRC, Hitchhiker
  - Efficient repair approaches for general erasure codes
    - e.g., lazy repair, PPR

Conventional Repair

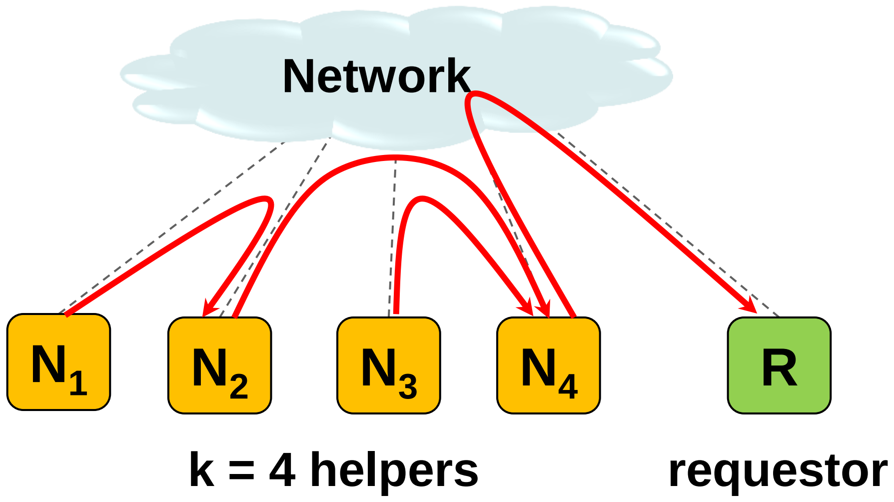
- Single-block repair:
  - Retrieve k blocks from k working nodes (helpers)
  - Store the repaired block at requestor



- Repair time = **k** timeslots
  - Bottlenecked by requestor's downlink
  - Uneven bandwidth usage (e.g., links among helpers are idle)

Partial-Parallel-Repair (PPR)

- Exploit linearity and addition associativity to perform repair in a "divide-and-conquer" manner



Timeslot 1:
N_1 sends a_1B_1 to N_2 → $a_1B_1+a_2B_2$
N_3 sends a_3B_3 to N_4 → $a_3B_3+a_4B_4$

Timeslot 2:
N_2 sends $a_1B_1+a_2B_2$ to N_4 → $a_1B_1+a_2B_2+a_3B_3+a_4B_4$

Timeslot 3:
N_4 → R → repaired block

- Repair time = **ceil($\log_2(k+1)$)** timeslots

Open Question

- Repair time of erasure coding remains larger than normal read time
  - Repair-optimal erasure codes still read more data than amount of failed data
- Erasure coding is mainly for warm/cold data
  - Repair penalty only applies to less frequently accessed data
  - Hot data remains replicated

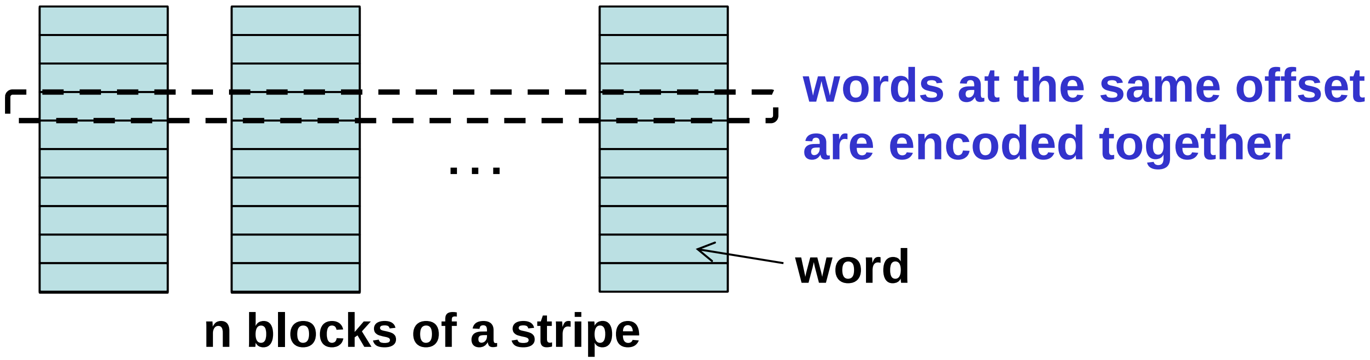
- **Can we reduce repair time of erasure coding to almost the same as the normal read time?**
  - Create opportunity for storing hot data with erasure coding

Our Contributions

- **Repair pipelining**, a technique to speed up repair for general erasure coding
  - Applicable for **degraded reads** and **full-node recovery**
  - **O(1) repair time** in homogeneous settings
- Extensions to heterogeneous settings
- A prototype ECPipe integrated with HDFS and QFS
- Experiments on local cluster and Amazon EC2
  - Reduction of repair time by 90% and 80% over conventional repair and partial-parallel-repair (PPR), respectively

Repair Pipelining

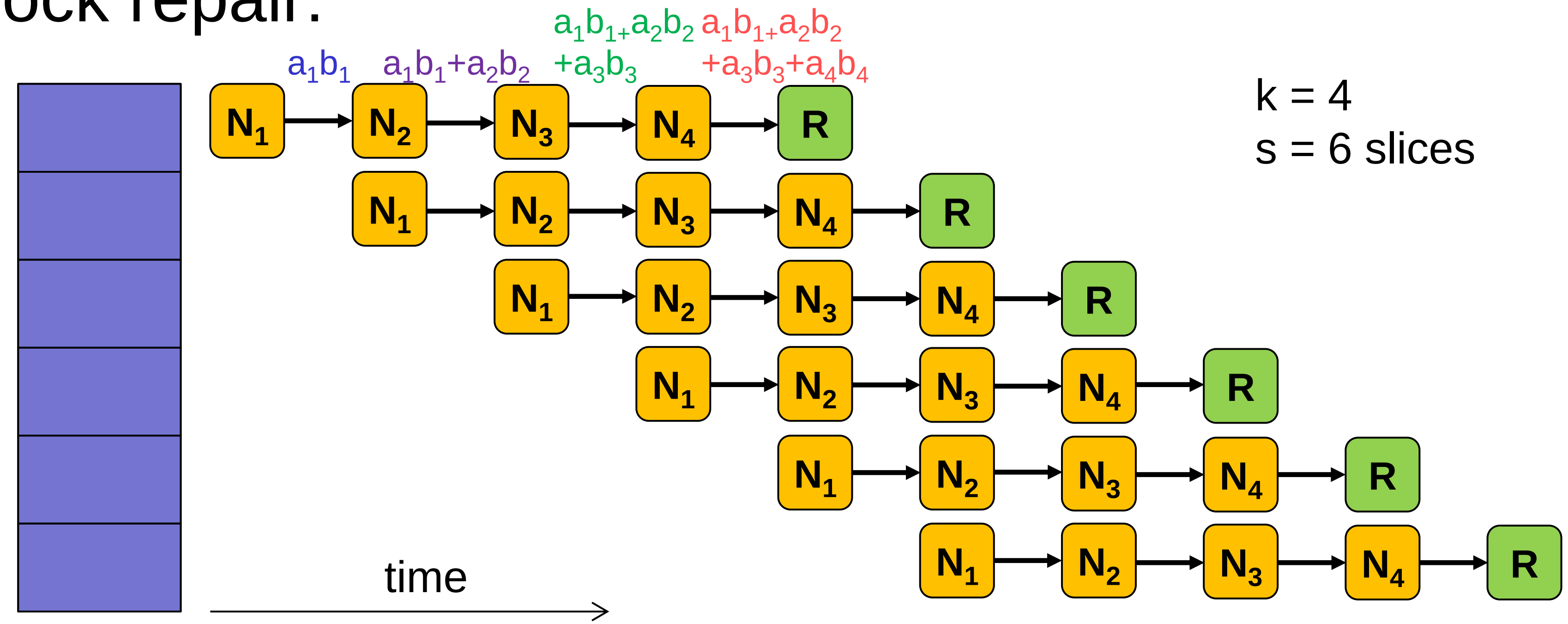
- **Goals**:
  - Eliminate bottlenecked links
  - Effectively utilize available bandwidth resources in repair
- **Key observation**: coding unit **(word)** is much smaller than read/write unit **(block)**
  - e.g., word size ~ 1 byte; block size ~ 64 MiB
  - Words at the same offset are encoded together in erasure coding



Repair Pipelining

- **Idea**: slicing a block
  - Each slice comprises multiple words (e.g., slice size ~ 32 KiB)
  - Pipeline the repair of each slice through a linear path

- Single-block repair:



k = 4
s = 6 slices

- Repair time = 1 + (k+1)/s → **1** timeslot if s is large

Repair Pipelining

- Two types of single-failure repair (most common case):
  - **Degraded read**
    - Repairing an unavailable block at a client
  - **Full-node recovery**
    - Repairing all lost blocks of a failed node at one or multiple nodes
    - Greedy scheduling of multiple stripes across helpers
- **Challenge**: repair degraded by **stragglers**
  - Any repair of erasure coding faces similar problems due to data retrievals from multiple helpers
- Our approach: address heterogeneity and bypass stragglers

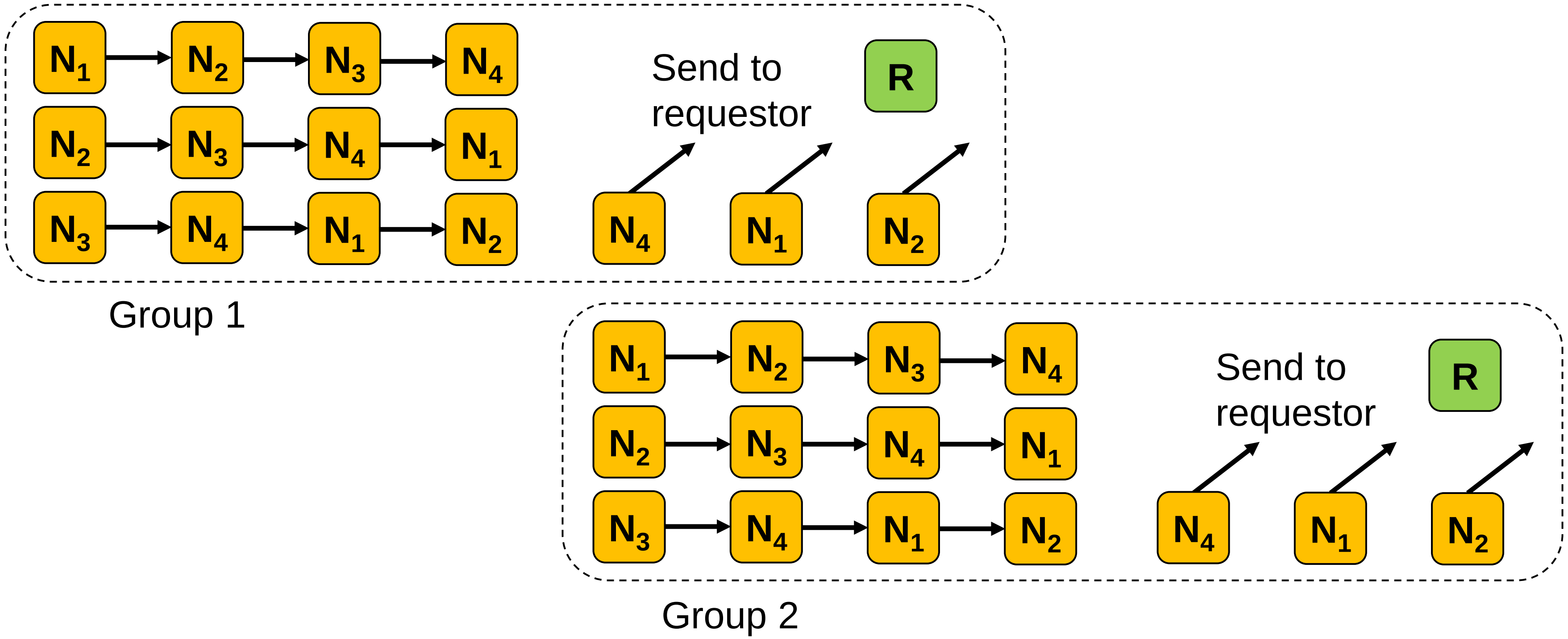
Extension to Heterogeneity

- Heterogeneity: link bandwidths are different

- Case 1: limited bandwidth when a client issues reads to a remote storage system
  - **Cyclic version of repair pipelining**: allow a client to issue parallel reads from multiple helpers

- Case 2: arbitrary link bandwidths
  - **Weighted path selection**: select the “best” path of helpers for repair

Repair Pipelining (Cyclic Version)



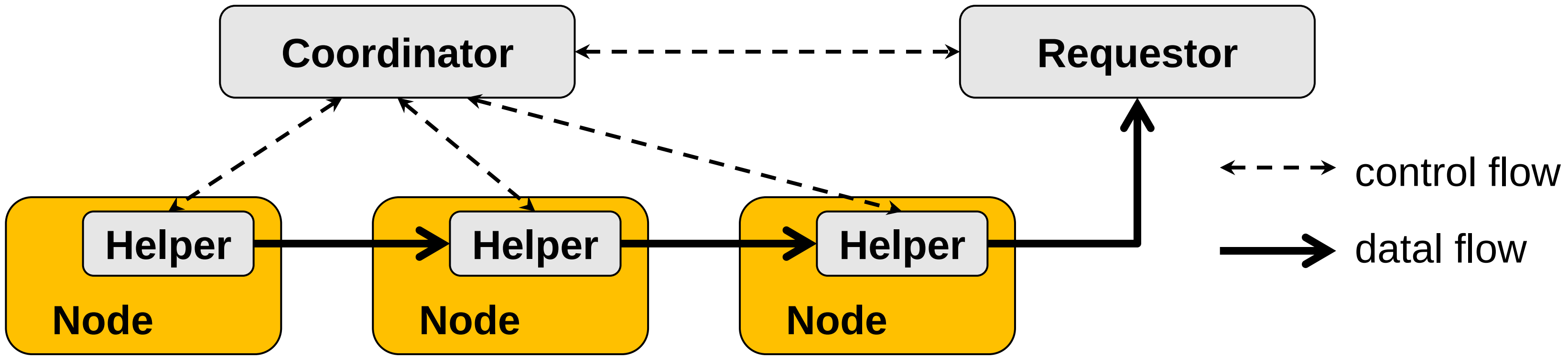
- Requestor receives repaired data from k-1 helpers
- Repair time in homogeneous environments → **1** timeslot for large s

Weighted Path Selection

- Goal: Find a path of k + 1 nodes (i.e., k helpers and requestor) that minimizes the maximum link weight
  - e.g., set link weight as inverse of link bandwidth
  - Any straggler is associated with large weight
- Brute-force search is expensive
  - (n-1)!/(n-1-k)! permutations
- Our algorithm:
  - Apply brute-force search, but avoid search of non-optimal paths
    - If link L has weight larger than the max weight of the current optimal path, any path containing L must be non-optimal
  - Remain optimal, with much less search time

Implementation

- **ECPipe**: a middleware atop distributed storage system

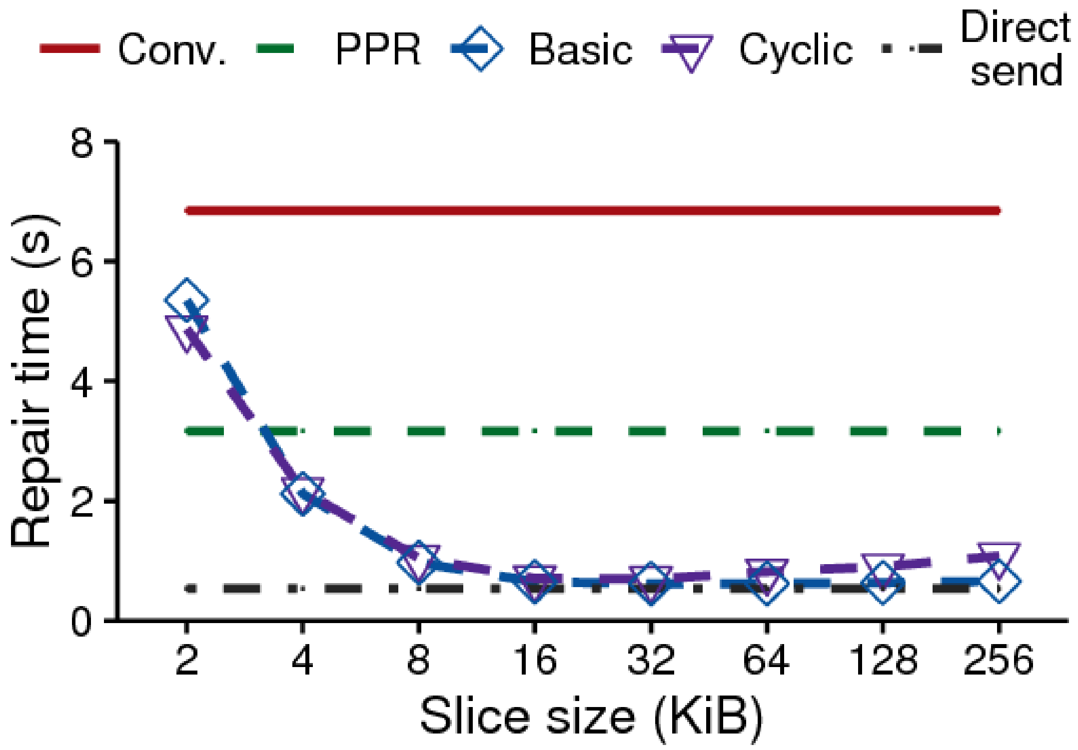


  - Requestor implemented as a C++/Java class
  - Each helper daemon directly reads local blocks via native FS
  - Coordinator access block locations and block-to-stripe mappings

- ECPipe is integrated with HDFS and QFS, with around 110 and 180 LOC of changes, respectively

Evaluation

➢ ECPipe performance on a 1Gb/s local cluster

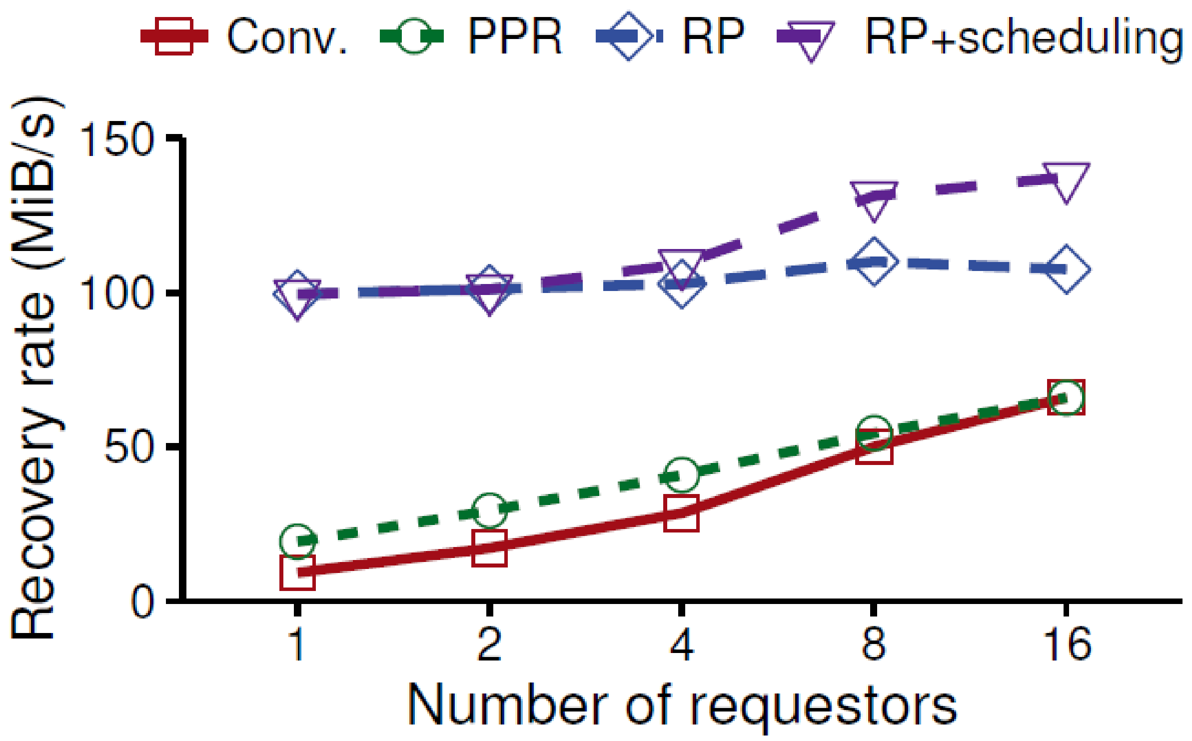


Single-block repair time vs. slice size for (n,k) = (14,10)

- ➢ Trade-off of slice size:
  - Too small: transmission overhead is significant
  - Too large: less parallelization
  - Best slice size = 32 KiB
- ➢ Repair pipelining (basic and cyclic) outperforms conventional and PPR by 90.9% and 80.4%, resp.
- ➢ Only 7% more than direct send time over a 1Gb/s link → O(1) repair time

Evaluation

➢ ECPipe performance on a 1Gb/s local cluster

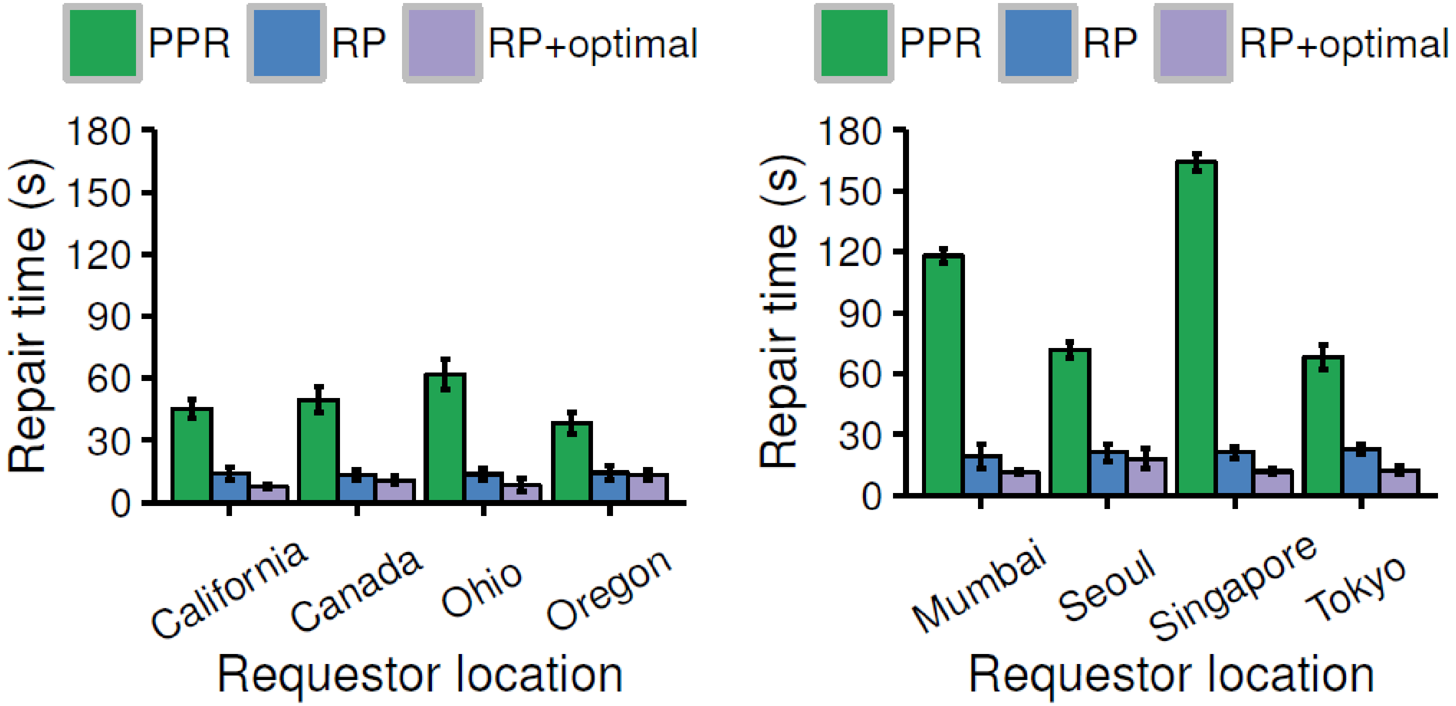


Full-node recovery rate vs. number of requestors for (n,k) = (14,10)

- ➢ Recovery rate increases with number of requestors
- ➢ Repair pipelining (RP and RP+scheduling) achieves high recovery rate
- ➢ Greedy scheduling balances repair load across helpers when there are more requestors (i.e., more resource contention)

Evaluation

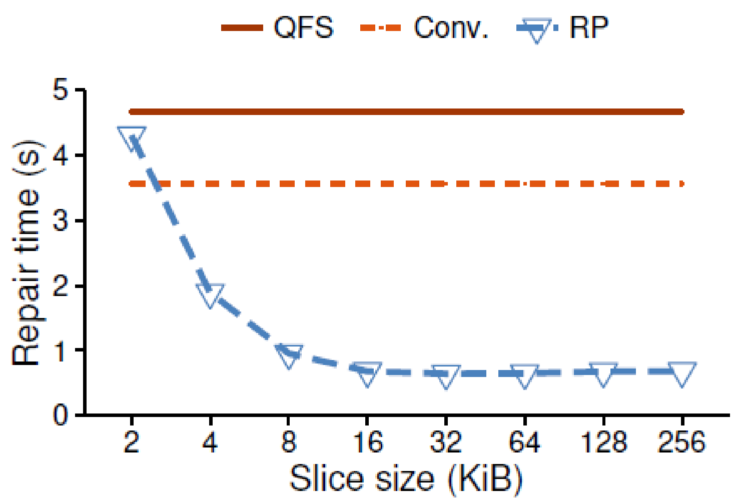
➢ **ECPipe performance on Amazon EC2**



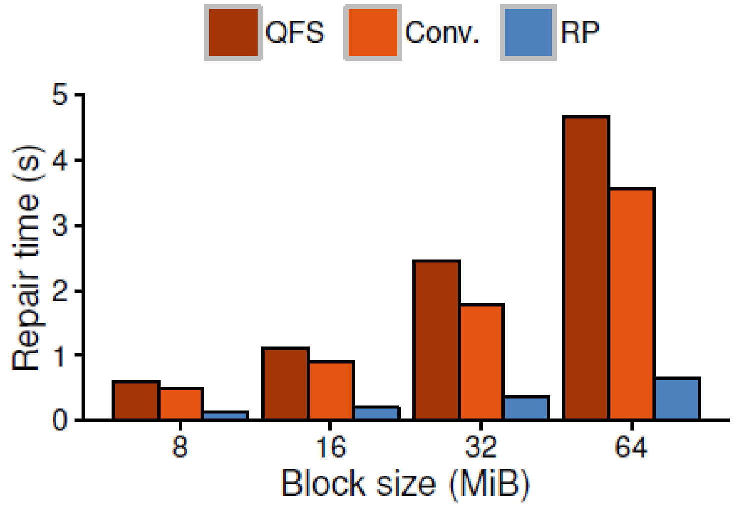
➢ Weighted path selection reduces single-block repair time of basic repair pipelining by up to 45%

Evaluation

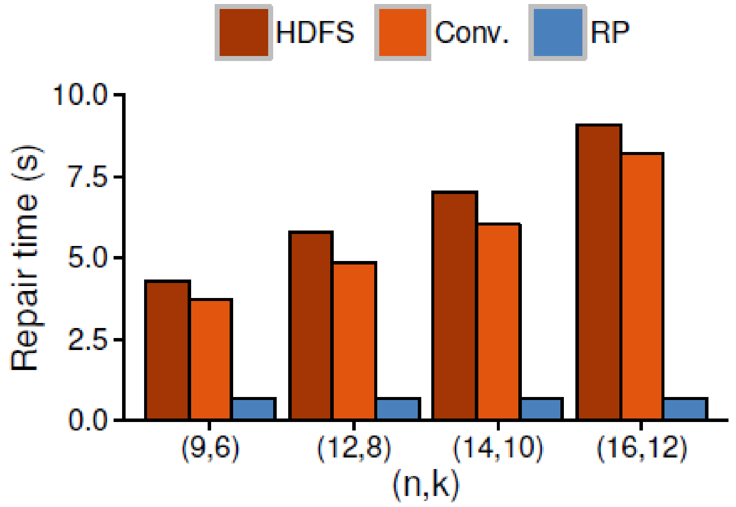
➢ Single-block repair performance on HDFS and QFS



QFS: slice size



QFS: block size



HDFS: (n,k)

➢ ECPipe significantly improves repair performance

- Conventional repair under ECPipe outperforms original conventional repair inside distributed file systems (by ~20%)
  - Avoid fetching blocks via distributed storage system routine
- Performance gain is mainly due to repair pipelining (by ~90%)

Conclusions

- Repair pipelining, a general technique that enables very fast repair for erasure-coded storage
- Contributions:
  - Designs for both degraded reads and full-node recovery
  - Extensions to heterogeneity
  - Prototype implementation ECPipe
  - Extensive experiments on local cluster and Amazon EC2
- Source code:
  - **http://adslab.cse.cuhk.edu.hk/software/ecpipe**