CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems

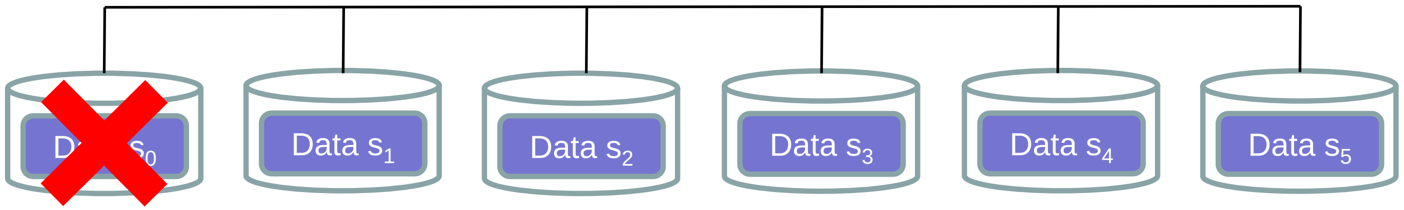
Runhui Li, **Jian Lin**, Patrick P. C. Lee

The Chinese University of Hong Kong

MSST'13

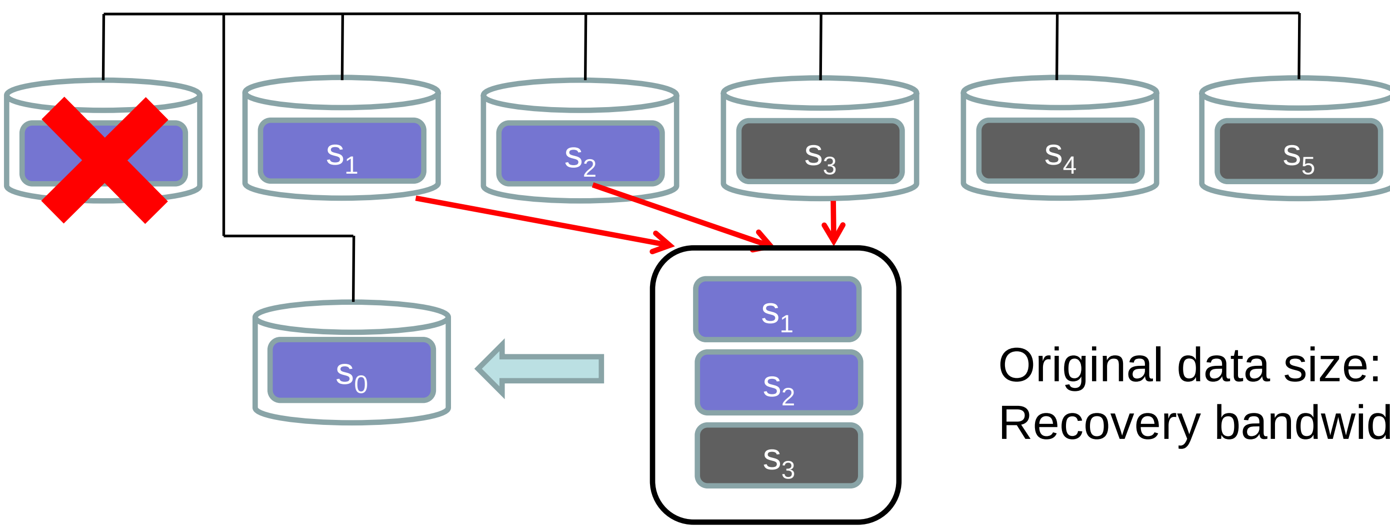
Motivation

- Large-scale distributed storage systems are widely used in enterprises (e.g., GFS, Azure)
- Data is distributed in a number of storage nodes
- Node failures are prevalent → data availability is critical



Erasure Codes

- Solution: add redundancy via erasure codes
- Example: (6, 3)-Reed-Solomon code



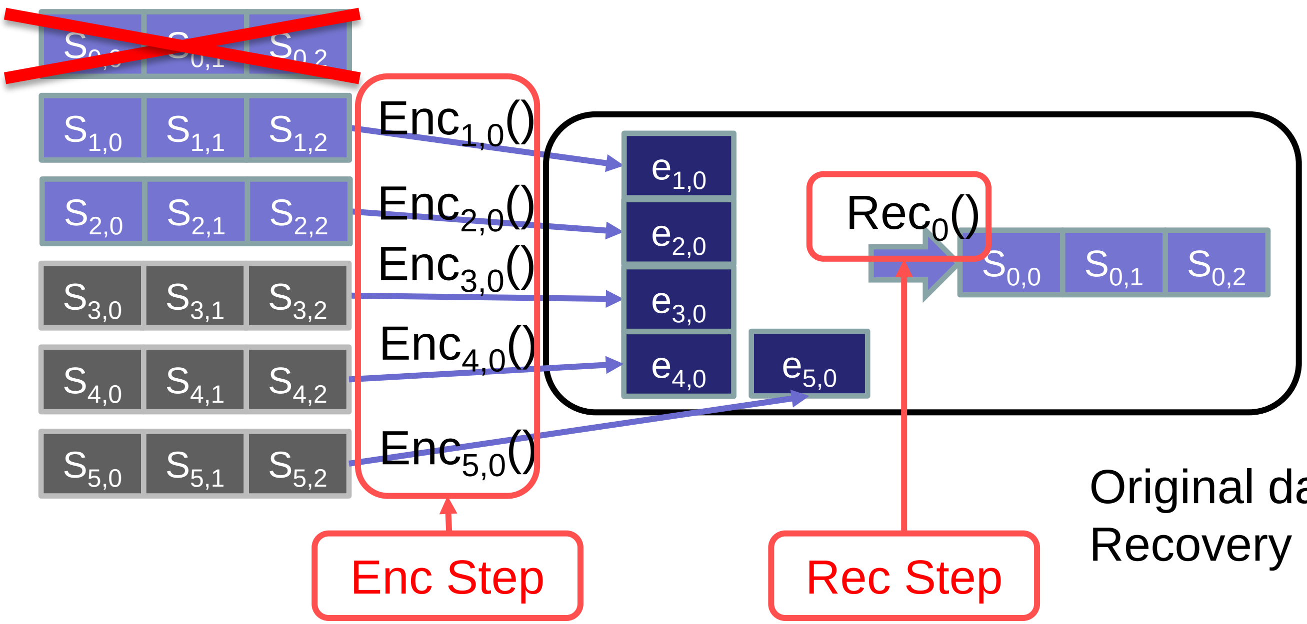
Original data size: M
Recovery bandwidth: M

- How to recover lost data?
  - **Recovery bandwidth**: amount of data downloaded from surviving nodes for recovery
  - Conventional approach reconstructs **all** original data to obtain lost data → High recovery bandwidth

Regenerating Codes

[Dimakis, ToIT'10]

➢ Minimize recovery bandwidth for a single node failure
   - **Enc** step: Every surviving node generates an <u>enc</u>oded symbol
   - **Rec** step: The newcomer <u>rec</u>onstructs the lost data with the encoded symbols



Original data size: M
Recovery bandwidth: 5M/9

Concurrent Node Failures

- Regenerating codes only designed for recovering a single node failure
- Correlated and co-occurring node failures are possible in practice:
  - In clustered storage systems [Schroeder, FAST'07; Ford, OSDI'10]
  - In dispersed storage systems [Chun NSDI'06; Shah NSDI'06]
- *Can we generalize existing regenerating codes to minimize recovery bandwidth for both single and concurrent failures?*

Related Work

- Cooperative recovery [Hu, JSAC'10; Kermarrec, NetCod'11]
  - Newcomers cooperate to reconstruct the lost data for multiple node failures
  - Implementation complexities unknown
- Minimizing recovery I/O [Khan, FAST'12; Huang, ATC'12]
  - Minimize the amount of disk read for single node failure recovery
  - Our work builds on regenerating codes that minimize recovery bandwidth

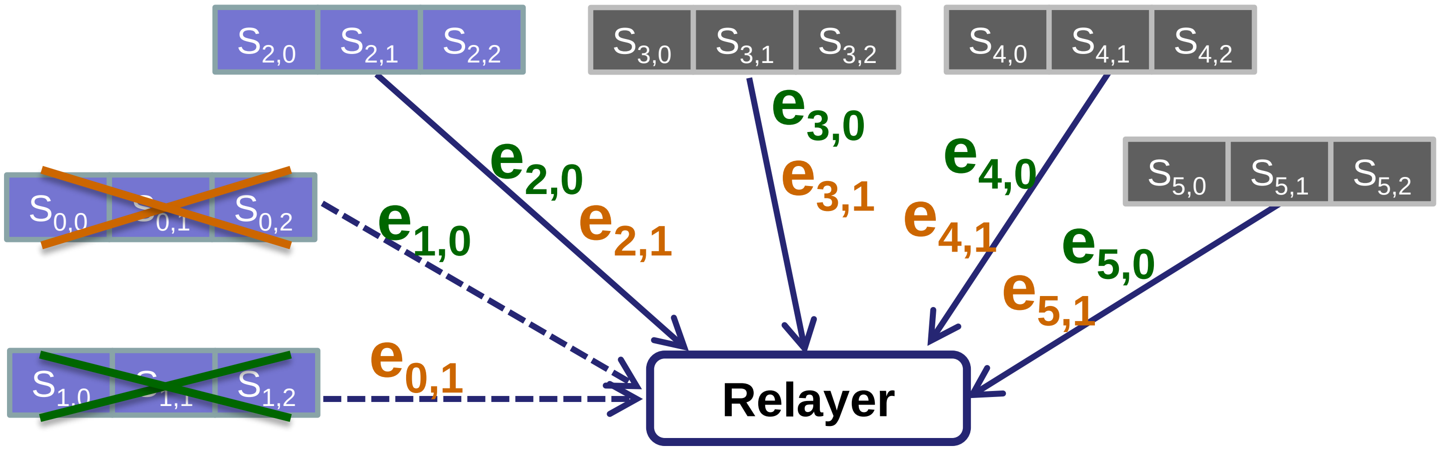
Our Work

- Build **CORE**, which augments existing optimized regenerating codes to support both single and **co**ncurrent failure **re**covery
  - Achieves minimum recovery bandwidth for concurrent failures in most cases
  - Retains existing optimal regenerating code constructions
- Implement CORE and evaluate our prototype atop a HDFS cluster testbed with up to 20 storage nodes

Main Idea

- Consider a system with n nodes
- Regenerating codes for single failure recovery:
  - Download one encoded symbol from each of n-1 surviving nodes
- CORE's idea for t-failure recovery (t > 1):
  - Treat t-1 failed nodes as logical surviving nodes
  - Reconstruct "virtual" symbols generated by the logical surviving nodes
  - Download real symbols from n-t surviving nodes
  - Reconstruct lost data of the remaining failed node

Example



$s_{0,0}, s_{0,1}, s_{0,2} = \mathrm{Rec}_0(e_{1,0}, e_{2,0}, e_{3,0}, e_{4,0}, e_{5,0})$

$e_{0,1} = \mathrm{Enc}_{0,1}(s_{0,0}, s_{0,1}, s_{0,2})$

$\quad = \mathrm{Enc}_{0,1}(\mathrm{Rec}_0(e_{1,0}, e_{2,0}, e_{3,0}, e_{4,0}, e_{5,0}))$

$s_{1,0}, s_{1,1}, s_{1,2} = \mathrm{Rec}_1(e_{0,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{5,1})$

$e_{1,0} = \mathrm{Enc}_{1,0}(s_{1,0}, s_{1,1}, s_{1,2})$

$\quad = \mathrm{Enc}_{1,0}(\mathrm{Rec}_1(e_{0,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{5,1}))$

Example

- We have two equations

  $$e_{0,1} = Enc_{0,1}(Rec_0(e_{1,0}, e_{2,0}, e_{3,0}, e_{4,0}, e_{5,0}))$$

  $$e_{1,0} = Enc_{1,0}(Rec_1(e_{0,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{5,1}))$$

- Trick: They form a linear system of equations
- If the equations are linearly independent, we can calculate $e_{0,1}$ and $e_{1,0}$
- Then we obtain lost data by

  $$s_{0,0}, s_{0,1}, s_{0,2} = Rec_0(e_{1,0}, e_{2,0}, e_{3,0}, e_{4,0}, e_{5,0})$$

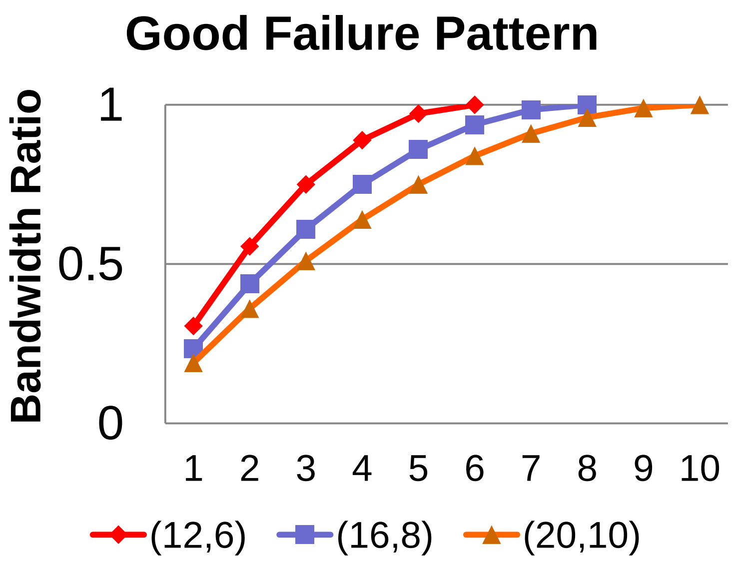
  $$s_{1,0}, s_{1,1}, s_{1,2} = Rec_1(e_{0,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{5,1})$$

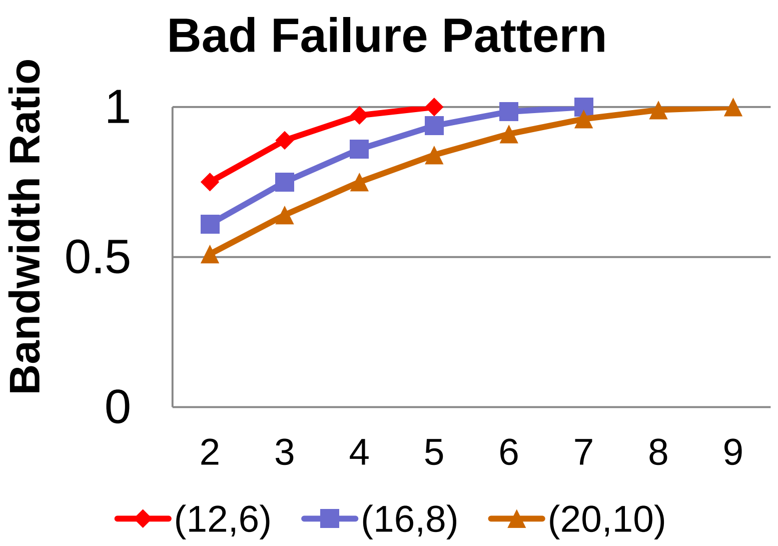
Bad Failure Pattern

- A system of equations may not have a unique solution. We call this a bad failure pattern
- Bad failure patterns count for less than ~1%
- Our idea: reconstruct data by adding one more node to bypass the bad failure pattern
  - Suppose nodes 0,1 form a bad failure pattern and nodes 0,1,2 form a good failure pattern. Reconstruct lost data for nodes 0,1,2
  - Still achieve bandwidth saving over conventional

Bandwidth Saving

- **Bandwidth Ratio**: Ratio of CORE to conventional in recovery bandwidth





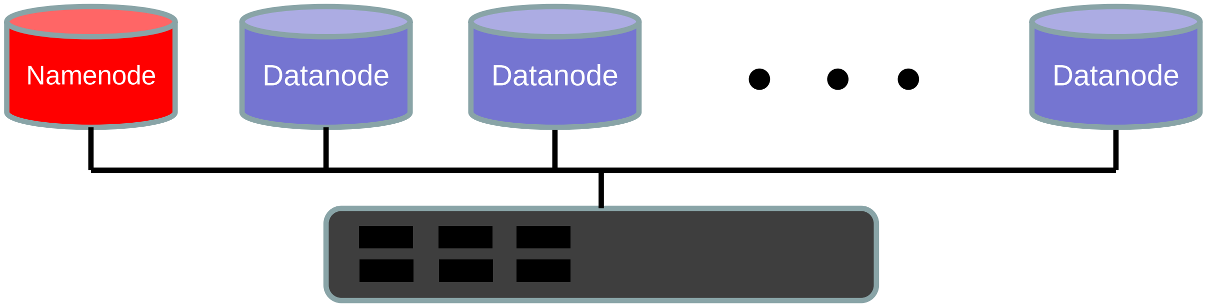
- Bandwidth saving of CORE is significant
  - e.g., (20,10)
  - Single failure: ~80%
  - 2-4 concurrent failures: 36-64%

Theorem

- **Theorem:** CORE, which builds on regenerating codes for single failure recovery, achieves the lower bound of recovery bandwidth if we recover a good failure pattern with $t \geq 1$ failed nodes
  - Over ~99% of failure patterns are good
- Proof in technical report

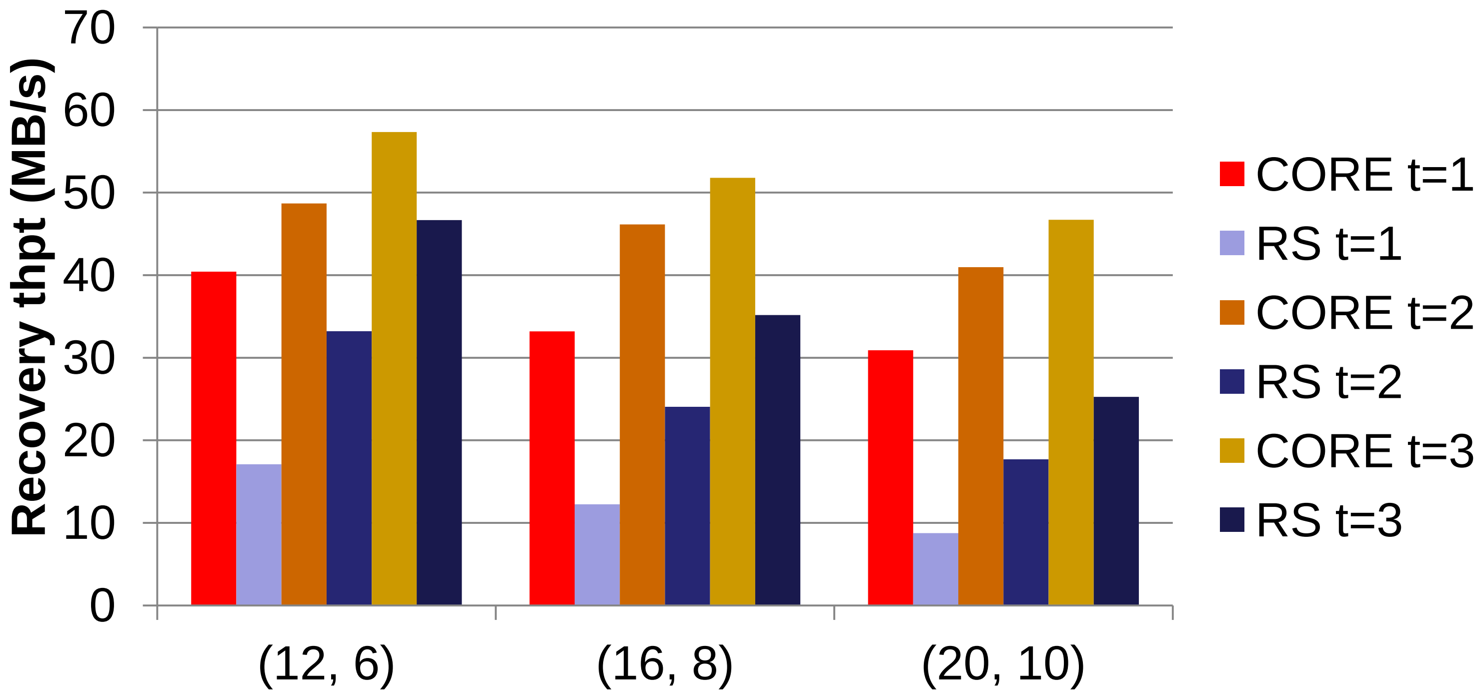
Experiments

- CORE built on HDFS
- Testbed:
  - 1 namenode, and up to 20 datanodes
  - Quad core 3.1GHz CPU, 8GB RAM, 7200RPM SATA harddisk, 1Gbps Ethernet



- Coding schemes:
  - Reed-Solomon codes vs. CORE (interference alignment codes)
- Metric:
  - Recovery throughput: lost data size / recovery time

Recovery Throughput



- CORE shows significantly higher throughput
  - e.g., in (20, 10), for single failure, the gain is **3.45x**; for two failures, it's **2.33x**; for three failures, is **1.75x**

Conclusions

- Build CORE to augment regenerating codes for concurrent failure recovery
  - Achieve minimum recovery bandwidth for most cases
- Implement CORE and integrate with HDFS
- Show via testbed experiments that CORE achieves higher recovery throughput over conventional recovery
- Source code of CORE is available at:
  - http://ansrlab.cse.cuhk.edu.hk/software/core/