Effective Replica Maintenance for Distributed Storage Systems

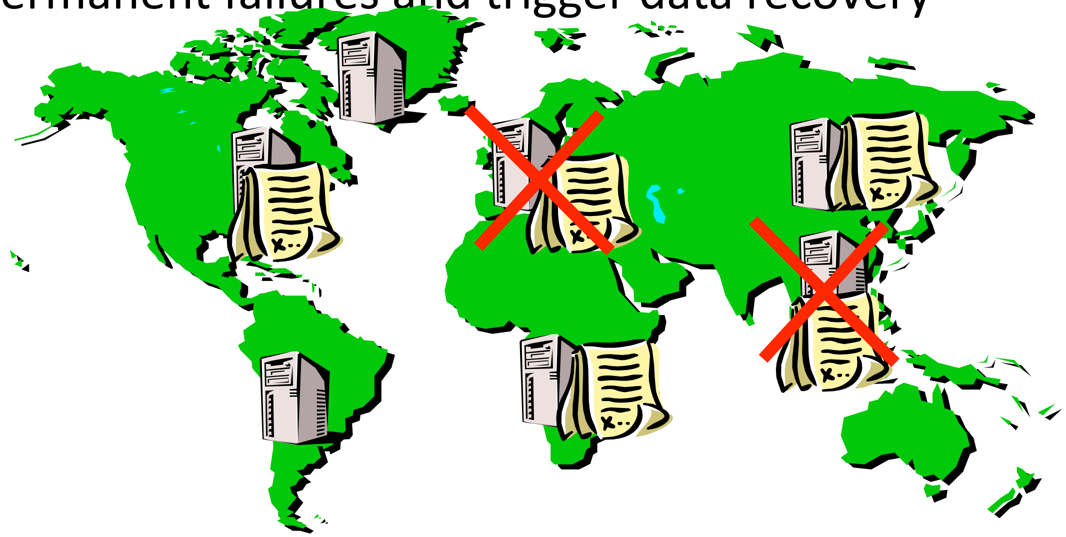
USENIX NSDI2006

Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon,
M. Frans Kaashoek, John Kubiatowicz, and Robert Morris

Presenter: Hakim Weatherspoon

Motivation

- Efficiently Maintain Wide-area Distributed Storage Systems
- Redundancy
  - duplicate data to protect against data loss
- Place data throughout wide area
  - Data availability and durability
- Continuously repair loss redundancy as needed
  - Detect permanent failures and trigger data recovery



Motivation

- **Distributed Storage System**: a network file system whose storage nodes are **dispersed over the Internet**
- **Durability**: objects that an application has put into the system are **not lost** due to disk failure
- **Availability**: get request will be able to return the object **promptly**

Motivation

- To store immutable objects **durably** at a **low bandwidth cost** in a distributed storage system

Contributions

- A set of techniques that allow wide-area systems to efficiently **store** and **maintain** large amounts of data
- An implementation: Carbonite

Outline

- Motivation
- **Understanding durability**
- Improving repair time
- Reducing transient failure cost
- Implementation Issues
- Conclusion

Providing Durability

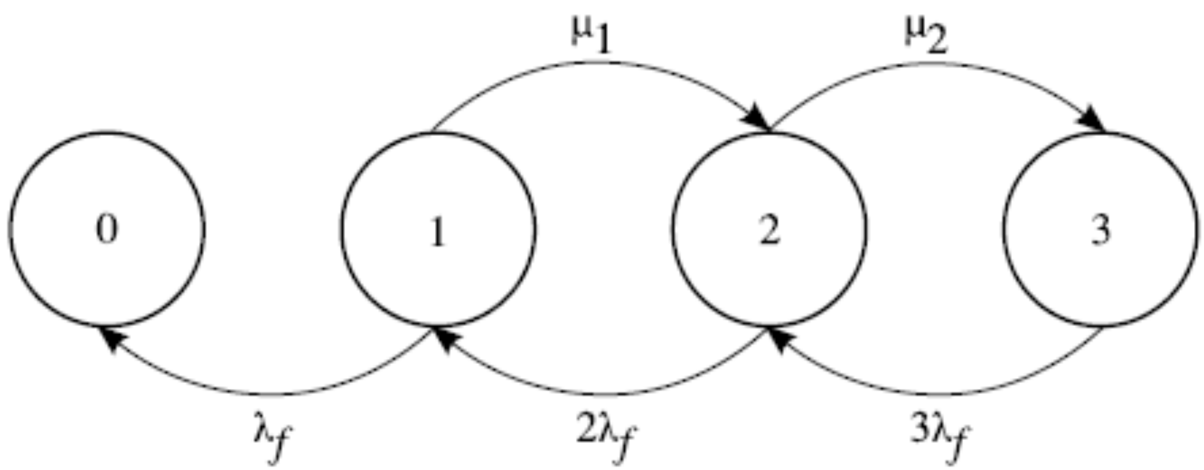
- Durability is relatively more important than availability
- Challenges
  - Replication algorithm: Create new replica faster than losing them
  - Reducing network bandwidth
  - Distinguish transient failures from permanent disk failures
  - Reintegration

Challenges to Durability

- Create new replicas **faster** than replicas are destroyed
  - Creation rate < failure rate ➔ system is **infeasible**
    - Higher number of replicas do not allow system to survive a higher average failure rate
  - Creation rate = failure rate + ε (ε is small) ➔ burst of failure may destroy all of the replicas

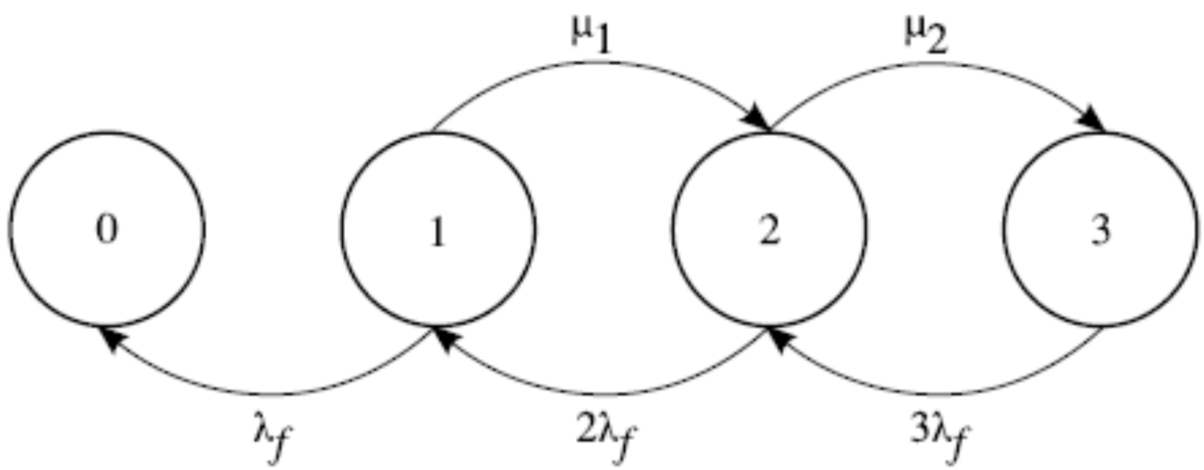
Number of Replicas as a Birth-Death Process

- Assumption: independent exponential inter-failure and inter-repair times
- λ_f : average failure rate
- μ_i: average repair rate at state i
- r_L : lower bound of number of replicas (r_L = 3 in this case)



Model Simplification

- Fixed μ and λ ➔ the equilibrium number of replicas is **$\Theta = \mu / \lambda$**
- If $\Theta < 1$, the system can no longer maintain full replication regardless of r_L



Real-world Settings

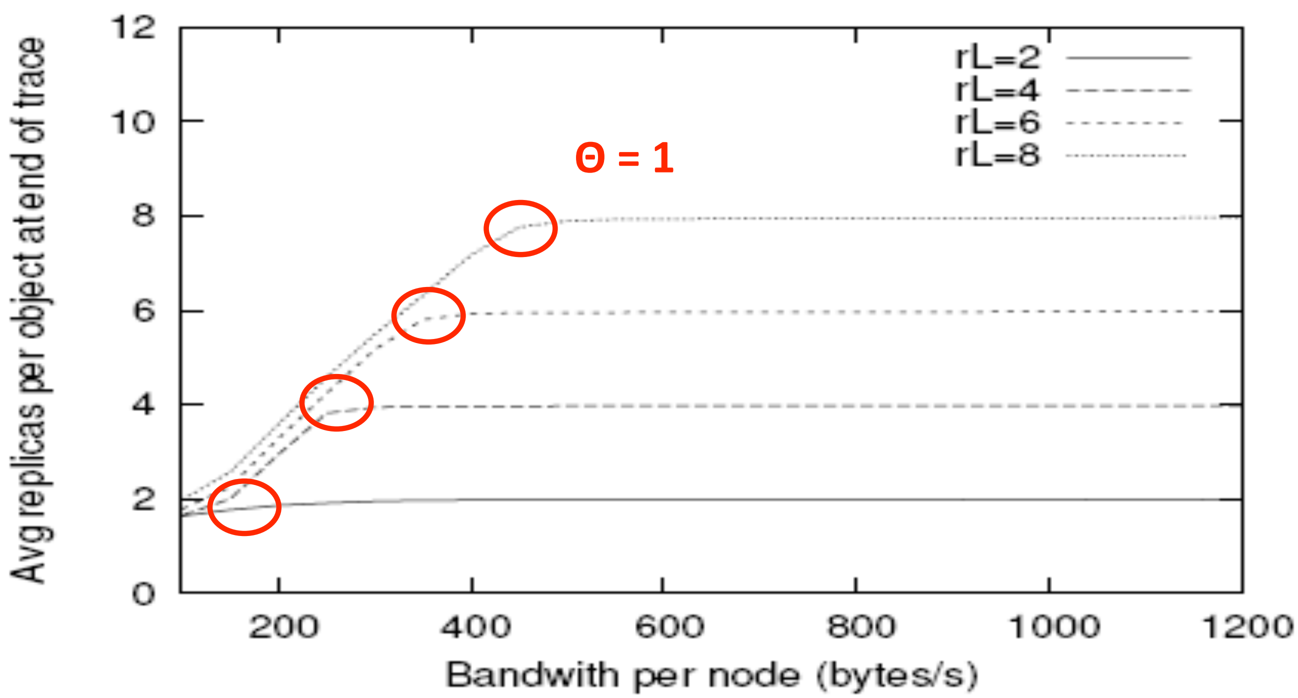
- Planetlab
  - 490 nodes
  - Average inter-failure time 39.85 hours
  - 150 KB/s bandwidth
- Assumption
  - 500 GB per node
  - $r_L = 3$

- λ = 365 day / 490 x (39.85 / 24) = 0.439 disk failures / year
- μ = 365 day / (500 GB x 3 / 150 KB/sec) = 3 disk copies / year

- $\Theta = \mu / \lambda = 6.85$

Impact of Θ

- Θ is the **theoretical upper limit** of replica number
- bandwidth ↑ ➔ μ ↑ ➔ Θ ↑
- r_L ↑ ➔ μ ↓ ➔ Θ ↓



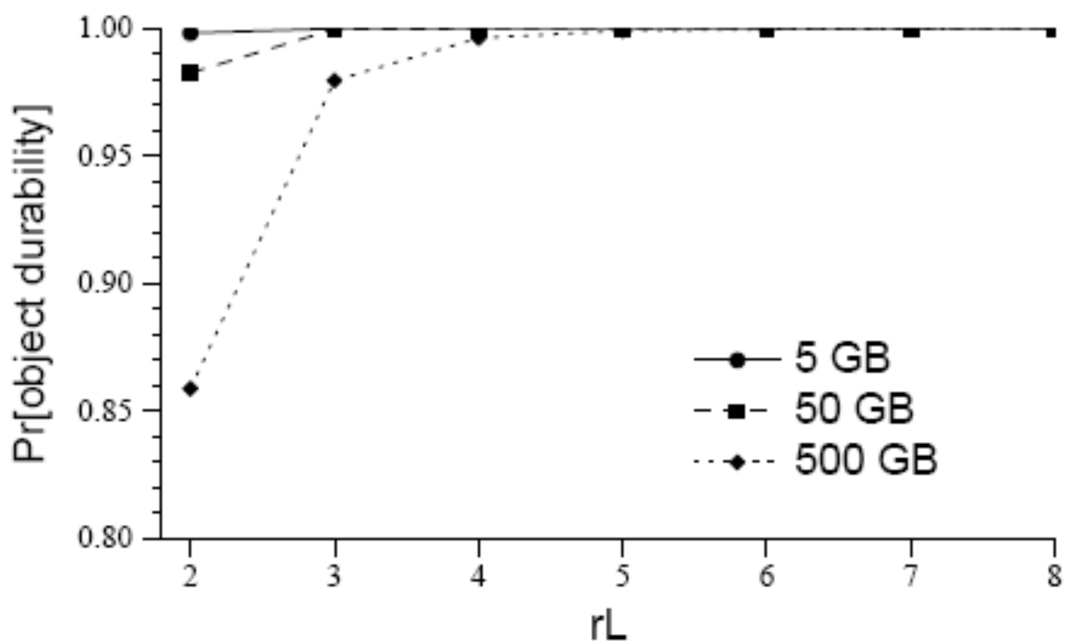
Choosing r_L

- Guidelines
  - Large enough to ensure durability
  - At least one more than the maximum burst of simultaneous failures
  - Small enough to ensure $r_L \leq \Theta$

r_L vs Durablility

- Higher r_L would cost high but tolerate more burst failures
- Larger data size ➔ $\lambda \uparrow$ ➔ need higher r_L

Analytical results from Planetlab traces (4 years)



Outline

- Motivation
- Understanding durability
- **Improving repair time**
- Reducing transient failure cost
- Implementation Issues
- Conclusion

Definition: Scope

- Each node, n, designates a **set** of other nodes that can potentially hold copies of the objects that n is responsible for. We call the **size** of that set the node's **scope**.
- scope $\in [r_L, N]$
  - N: number of nodes in the system

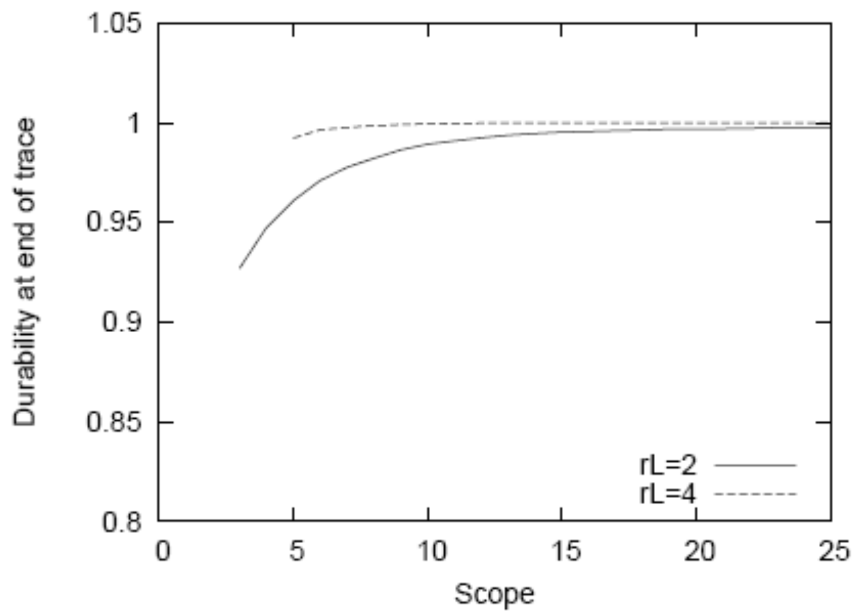
Effect of Scope

- Small scope
  - Easy to keep track of objects
  - More effort of creating new objects

- Big scope
  - **Reduces repair time**, thus increases durability
  - Need to monitor many nodes
  - If large number of objects and random placement, may increase the likelihood of simultaneous failures

Scope vs. Repair Time

- Scope ↑ ➔ repair work is spread over more access links and completes faster
- r_L ↓ ➔ scope must be higher to achieve the same durability



Outline

- Motivation
- Understanding durability
- Improving repair time
- **Reducing transient failure cost**
- Implementation Issues
- Conclusion

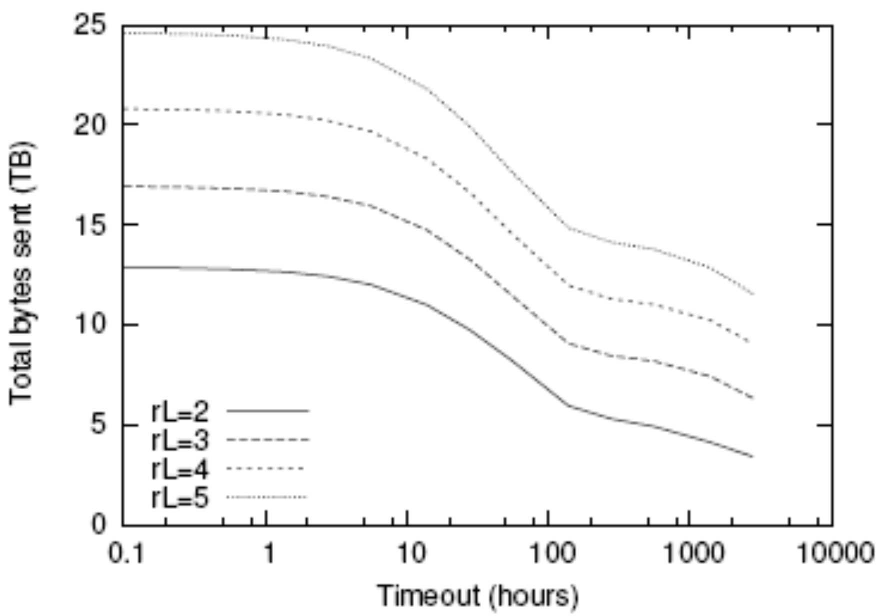
The Reasons

- Not creating new replicas for transient failures
  - Unnecessary costs (replicas)
  - Waste resources (bandwidth, disk)

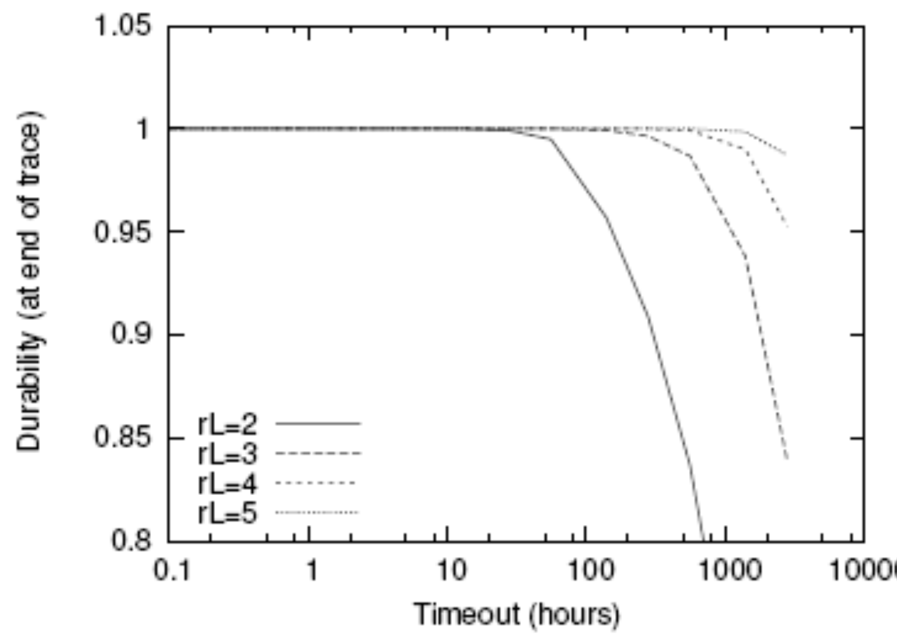
- Solutions
  - **Timeouts**
  - **Reintegration**
  - **Batch**
  - **Erasure codes**

Timeouts

- Timeout > average down time
  - Average down time: 29 hours
  - Reduce maintenance cost
  - Durability still maintained



- Timeout >> average down time
  - Durability begins to fall
  - Delays the point at which the system can begin repair



Reintegration

- Reintegrate replicas stored on nodes after transient failures
- System must be able to track more than rL number of replicas
- Depends on **a**: the average fraction of time that a node is available

Effect of Node Availability

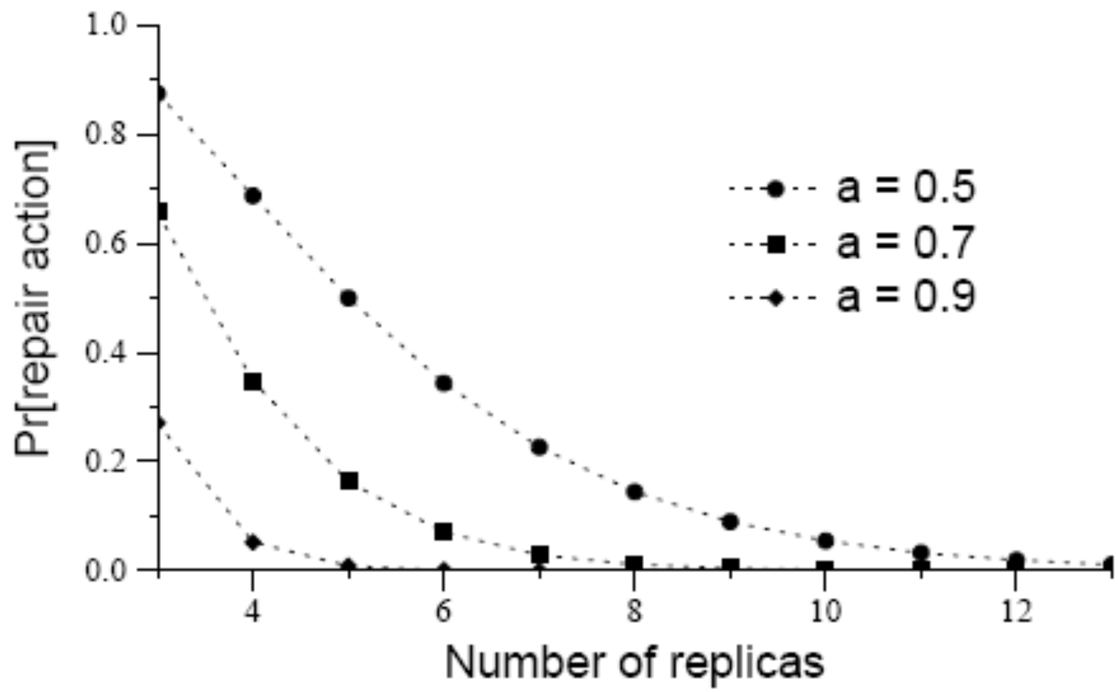
- Pr[new replica needs to be created] == Pr[less than r_L replicas are available] :

$$\Pr[R < r_L \mid r \text{ extant copies}] = \sum_{i=0}^{r_L - 1} \binom{r}{i} a^i (1-a)^{r-i}.$$

- Chernoff bound: **$2r_L/a$** replicas are needed to keep r_L copies available

Node Availability vs. Reintegration

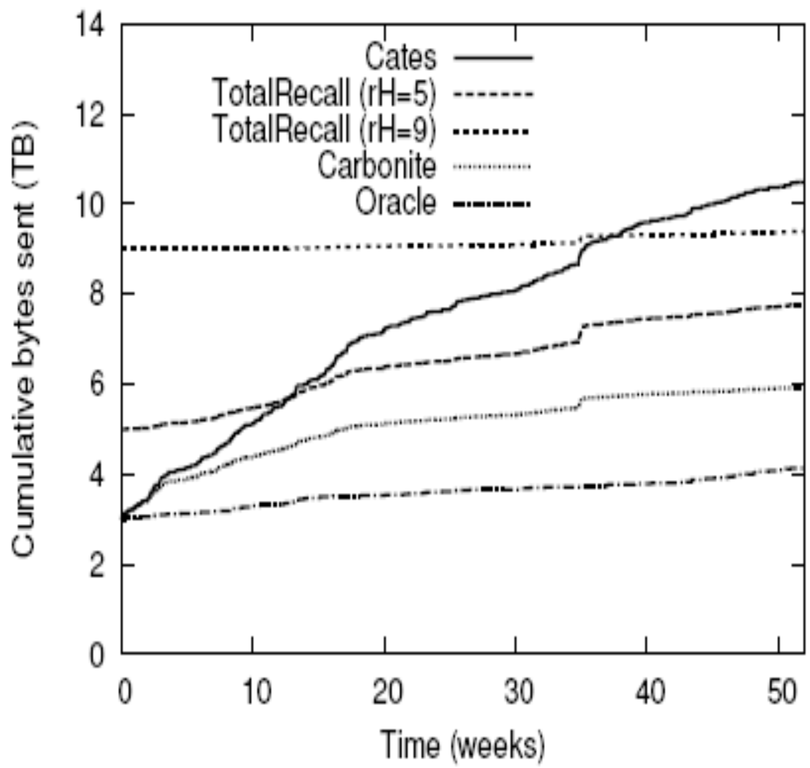
- Reintegrate can work safely with **$2r_L/a$** replicas
- $2/a$ is the **penalty** for not distinguishing transient and permanent failures
- $r_L = 3$

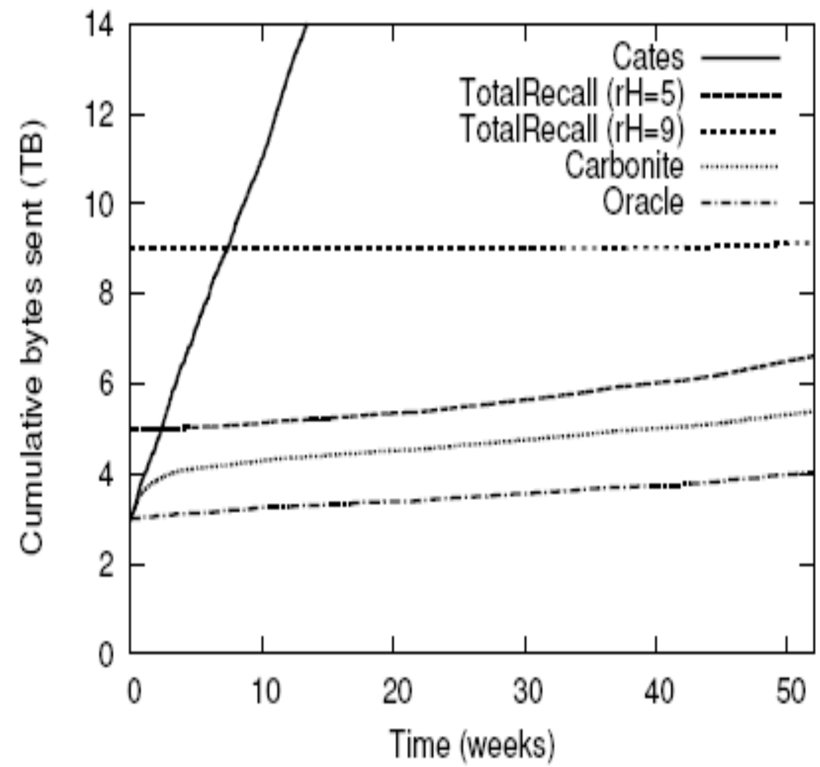


Four Replication Algorithms

- Cates
  - Fixed number of replicas r_L
  - Timeout
- Total Recall
  - Batch
- Carbonite
  - Timeout + reintegration
- Oracle
  - Hypothetical system that can differentiate transient failures from permanent failures

Effect of Reintegration

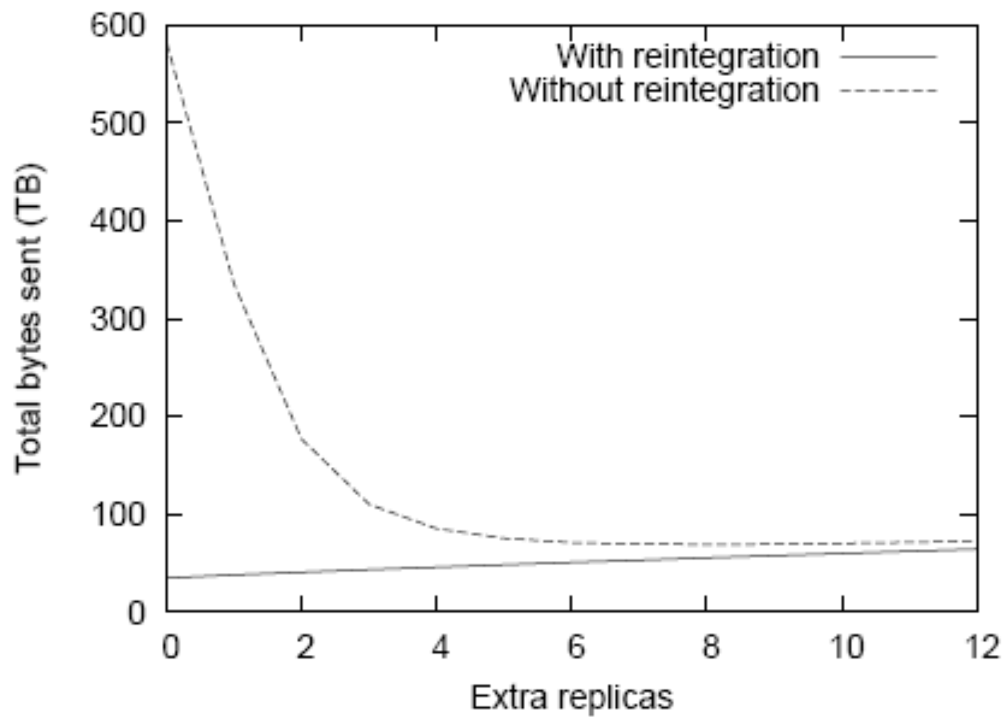




Batch

- In addition to r_L replicas, make **e** additional copies
  - Makes repair less frequent
  - Use up more resources

- $r_L = 3$

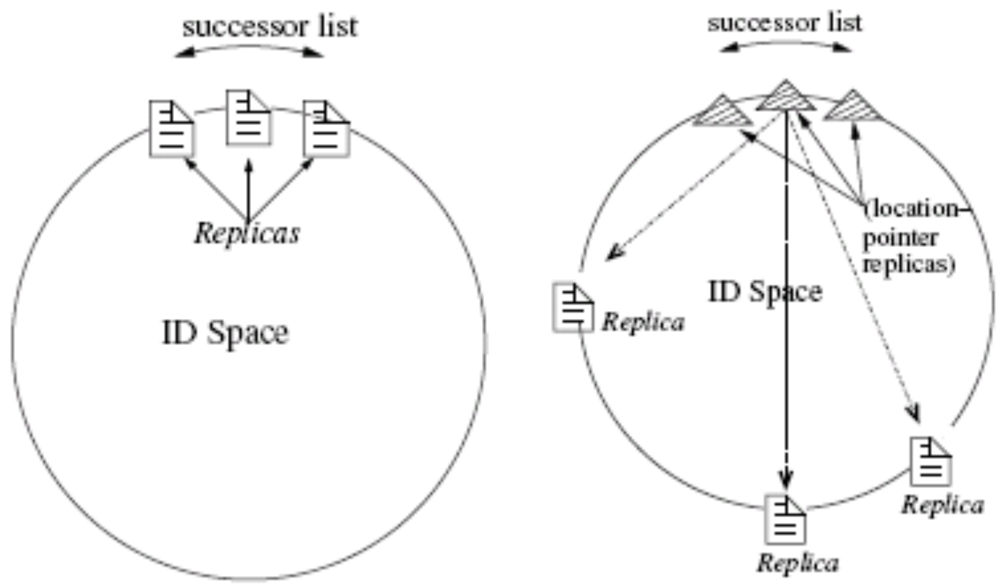


Outline

- Motivation
- Understanding durability
- Improving repair time
- Reducing transient failure cost
- **Implementation Issues**
- Conclusion

DHT vs. Directory-based Storage Systems

- DHT-based: consistent hashing an identifier space
- Directory-based : use indirection to maintain data, and DHT to store location pointers



(a) DHT (b) Directory

Node Monitoring for Failure Detection

- Carbonite requires that each node know the number of available replicas of each object for which it is responsible

- The goal of monitoring is to allow the nodes to track the number of available replicas

Monitoring consistent hashing systems

- Each node maintains, for each object, a **list of nodes in the scope** without a copy of the object

- When synchronizing, a node n provide key k to node n' who missed an object with key k, prevent n' from reporting what n already knew

Monitoring host availability

- DHT's routing tables uses the spanning tree rooted at each node a O(logN) out-degree
- Multicast **heartbeat** message of each node to its children nodes periodically
- When heartbeat is missed, monitoring node triggers repair actions

Outline

- Motivation
- Understanding durability
- Improving repair time
- Reducing transient failure cost
- Implementation Issues
- **Conclusion**

Conclusion

- Many design choices remain to be made
  - Number of replicas (depend on failure distribution and bandwidth, etc)
  - Scope size
  - Response to transient failures
    - Reintegration (extra copies #)
    - Timeouts (timeout period)
    - Batch (extra copies #)