Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility

Antony Rowstron and Peter Druschel

Presented by Anchal Agrawal

March 29, 2016

WHAT IS PAST?

PAST is a peer-to-peer storage system that offers caching and replication semantics in addition to P2P properties such as a self-organizing node overlay network.

KEY TAKEAWAYS

- Meant as an archival storage system, not a general-purpose file system
- File replication by storing a file at k nodes
- Support for caching popular files
- Storage management for nodes with various storage capacities and files of different sizes
- Uses Pastry for query routing and maintaining a node overlay network

PASTRY

Pastry is an efficient and self-organizing P2P routing protocol. Each node stores:

- A leaf set containing nodes with numerically closest IDs.
- A routing table with entries of nodeIDs with matching prefixes of the current nodeID.
- A neighborhood set containing nearby nodes. Used only during node addition and recovery.

Exchanging keep-alive messages helps detect node failures.

PASTRY BOOK-KEEPING

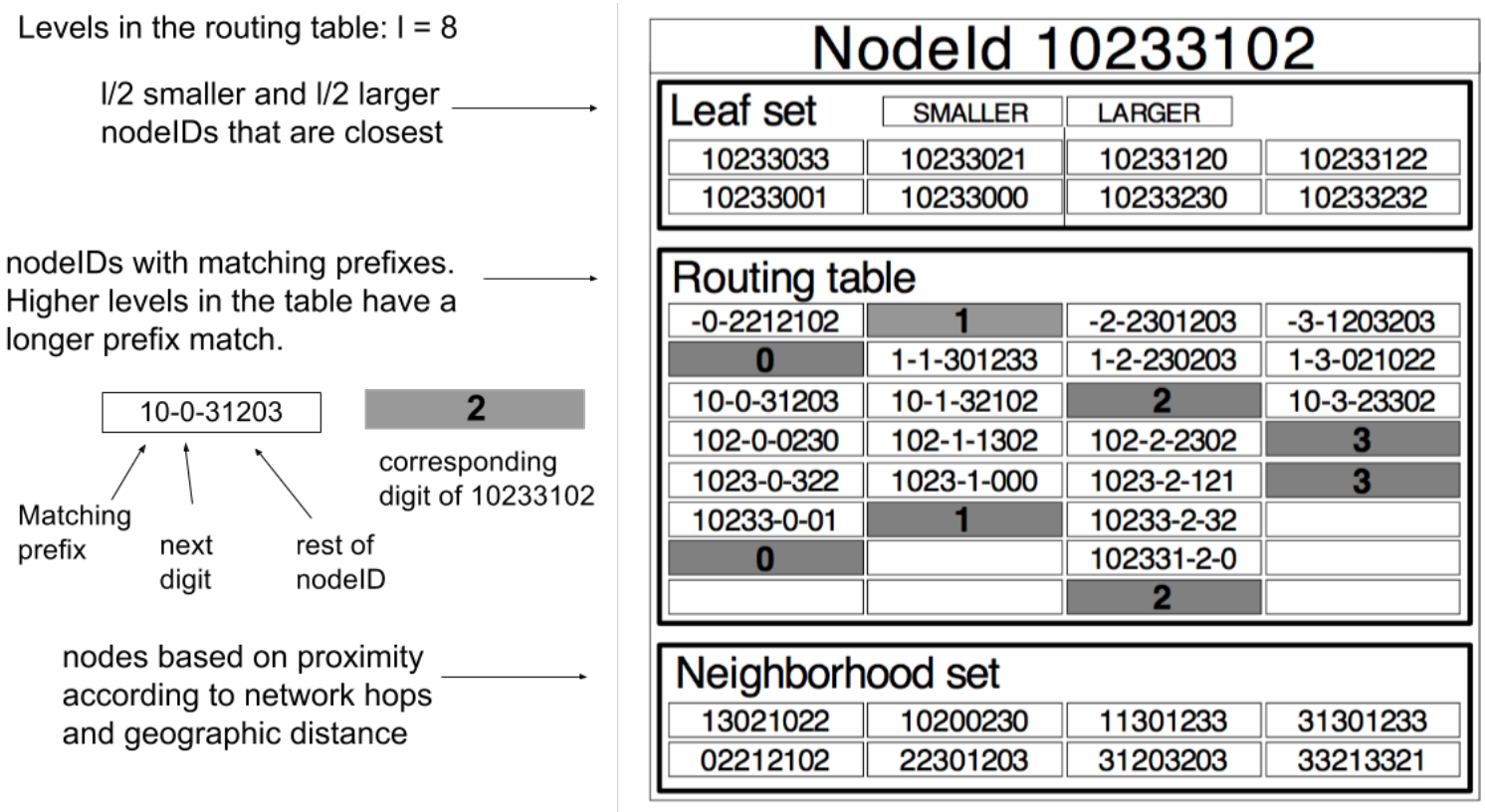

Figure 1: A sample routing table along with leaf and neighborhood sets.

PASTRY ROUTING

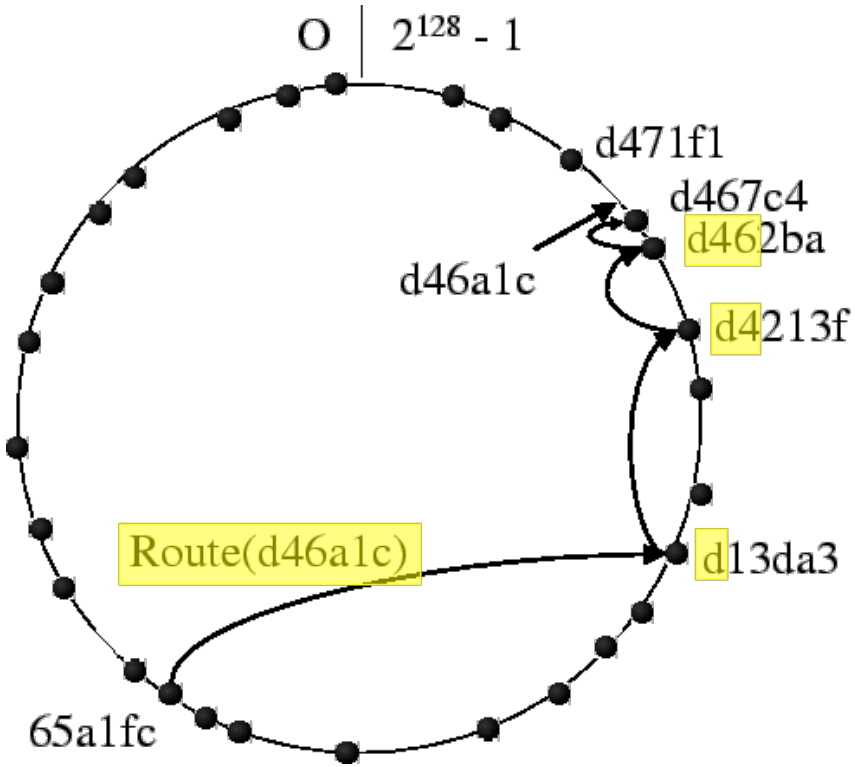

Figure 2: Routing a query with key d46a1c.

M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks.

PAST FILE OPERATIONS

- Insert

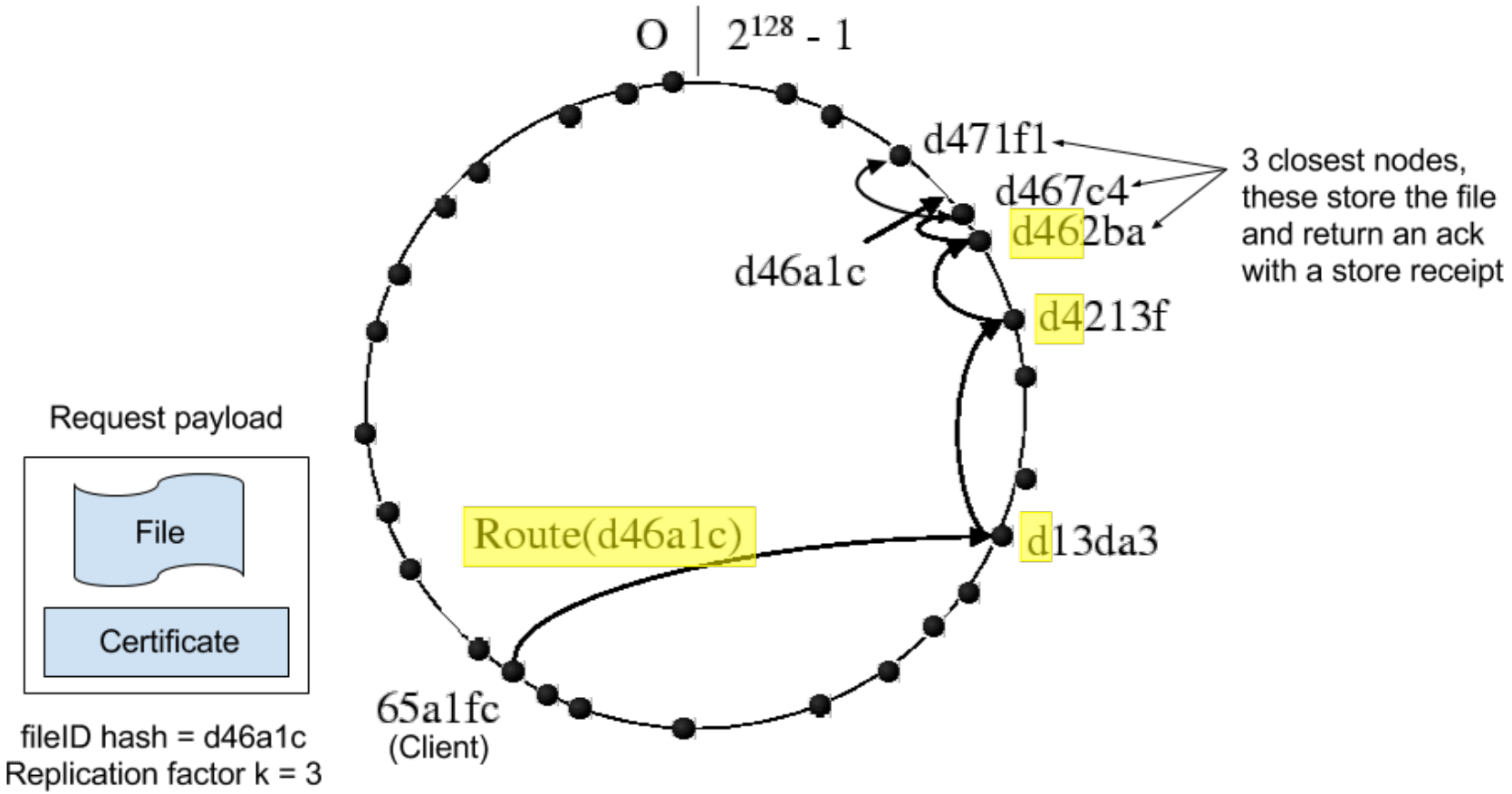

Figure 3: Inserting fileID d46a1c.

PAST FILE OPERATIONS

- Lookup
  - As soon the request reaches a node with the fileID, it is returned and the query is not forwarded.
- Reclaim ”weak delete”
  - The client sends a reclaim certificate which is used to verify that the owner is issuing the request.
  - After a reclaim, it is not guaranteed that a lookup will succeed and the file won’t exist in the system.

SECURITY

- Each node and user hold a *smartcard* associated with a public/private key.
- Store receipts and file/reclaim certificates ensure verification of operations.
- The system assumes that most nodes are well-behaved and an attacker can't control smartcards.
- Pastry's routing can be randomized to prevent malicious nodes from intercepting messages.

STORAGE MANAGEMENT

Two goals:

- Balance the remaining global storage as system utilization approaches 100%.
- Keep k replicas of files.

Not all k closest nodes may be able to accommodate a file due to:

- Different file sizes
- Different storage capacities of nodes
- Number of files assigned to a particular node

REPLICA DIVERSION

If a node A can't store a file, it picks a node B in its leaf set that's not among the k closest nodes and doesn't have a replica already. A then stores a pointer to the file on B.

What if A or B fails?

- If A fails, another node C which is the $k+1$th closest stores a file pointer to B.
- B's failure is handled by Pastry's node rejoining scheme.

REPLICA DIVERSION: EXAMPLE

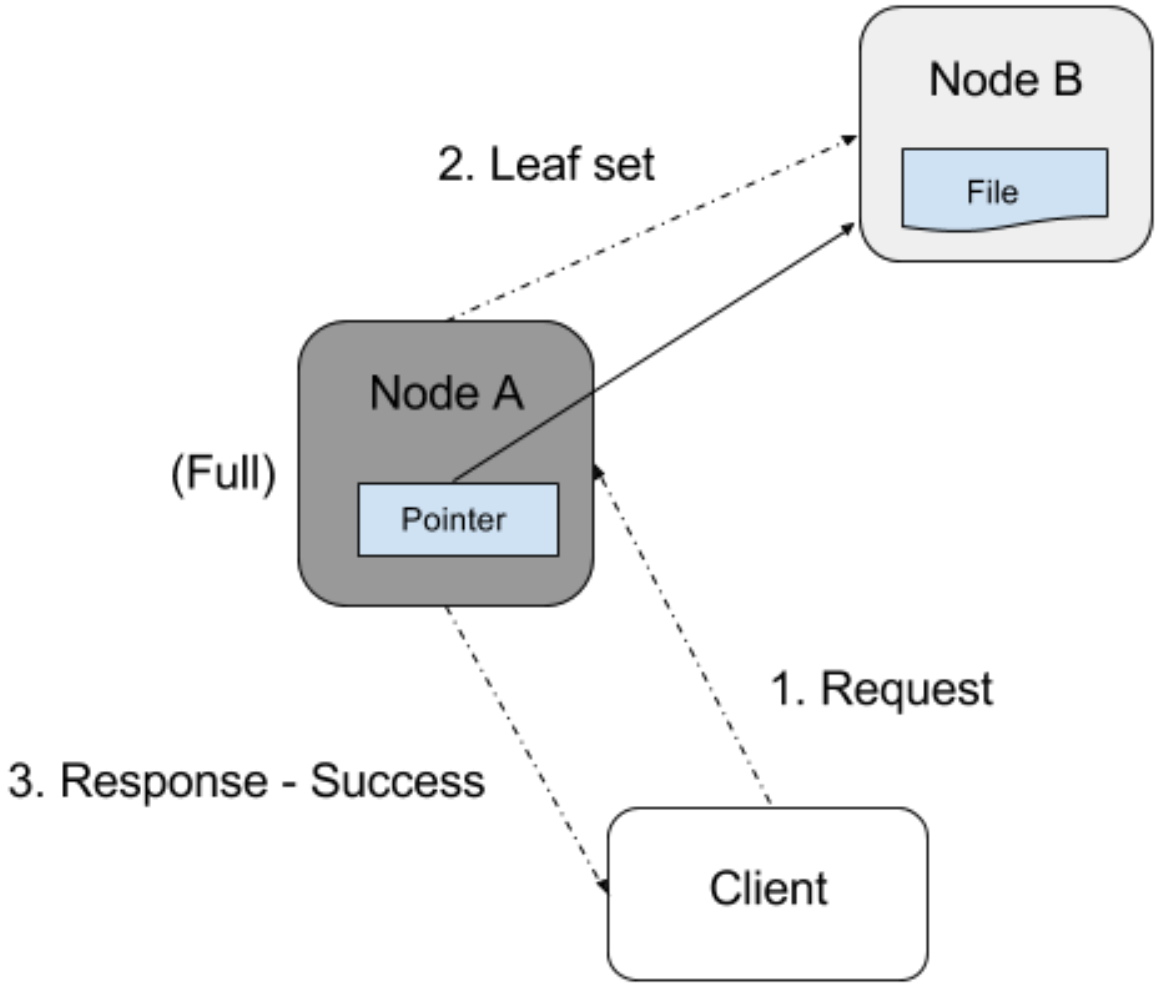

Figure 4: Replica diversion.

FAILURE OF NODE A

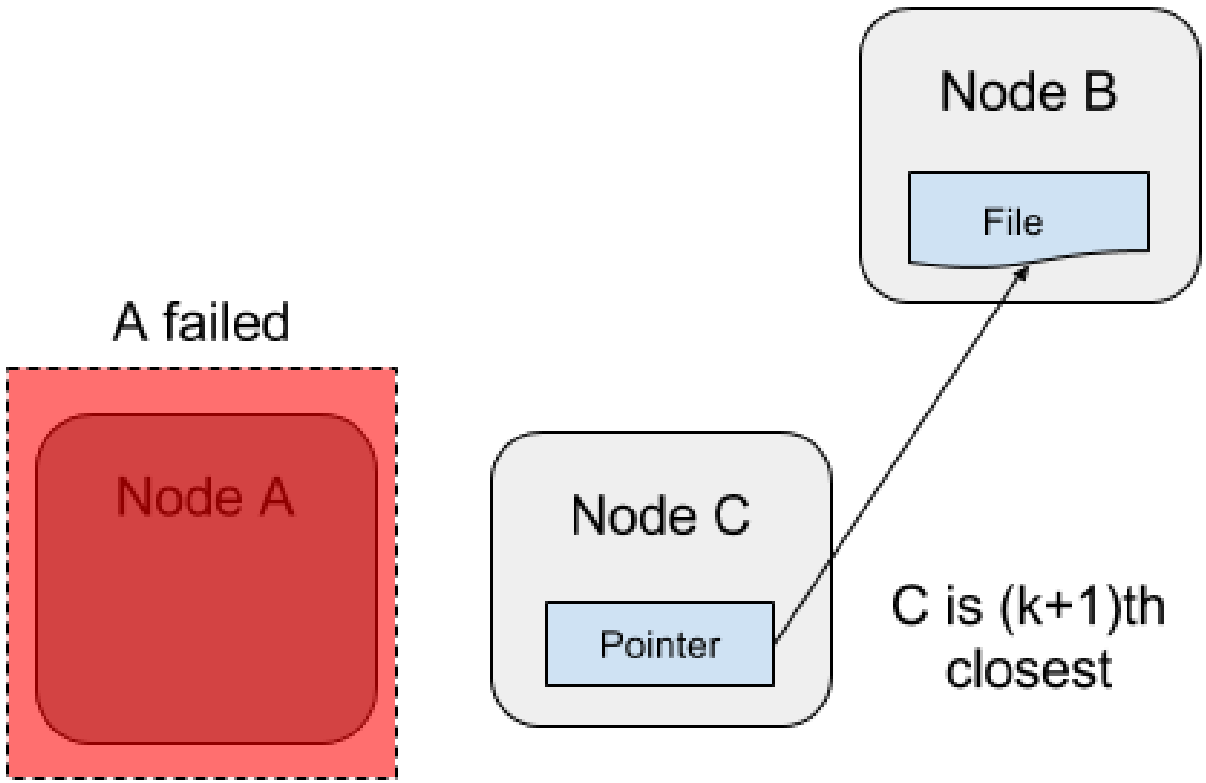

Figure 5: A fails after replica diversion.

FAILURE OF NODE B

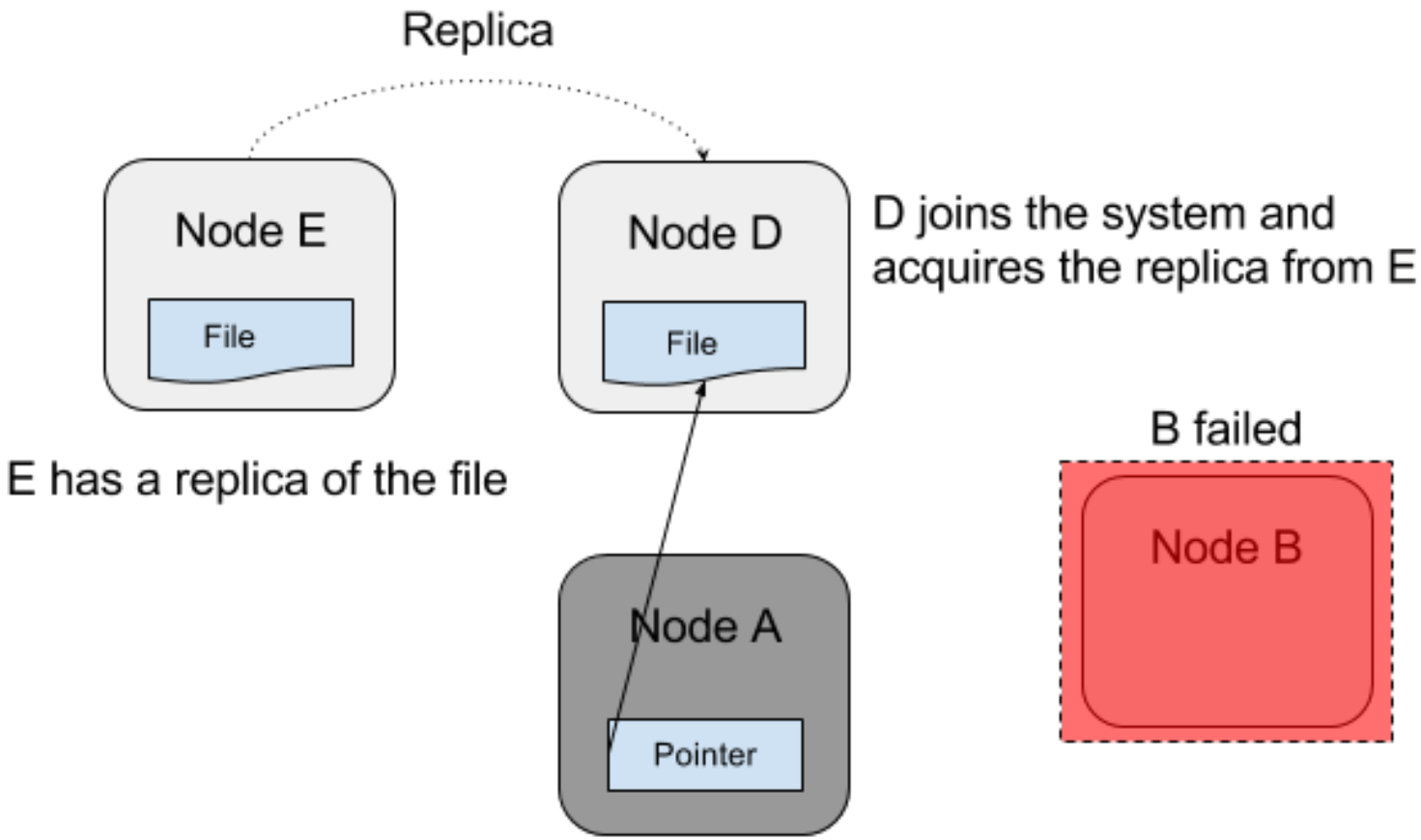

Figure 6: B fails after replica diversion.

REPLICA DIVERSION: POLICIES

To decide whether to accept a replica or not, nodes use a threshold

$$S_D/F_N$$

where $S_D :=$ file size and $F_N :=$ remaining storage on a node

Primary replica nodes use a threshold t_{pri} and secondary nodes use t_{div}, where $t_{pri} > t_{div}$. The size of an accepted file is large when t is large.

ANOTHER STRATEGY: FILE DIVERSION

If a node's leaf set is approaching full capacity, a different fileID is generated for the file by using a different salt. This directs the file to a different region of the node space.

HANDLING CHURN

A node joining the system may become one of the k closest nodes for certain files or may cease to be so. This creates overhead, which is avoided by storing pointers to files. The files are later transferred offline.

Leaf set changes are discovered by keep-alive messages and replicas are moved between nodes gradually.

CACHING

Lookups for popular files are optimized by storing them at more than k nodes. Files are cached at all nodes through which a lookup or insert is routed, if possible.

The GreedyDual-Size cache replacement policy is used, which uses weights to rank files. The weight is given by

$H_d = c(d)/s(d)$, where $c(d) :=$ cost of a file and $s(d) :=$ file size. The file with the minimum H_d is evicted when the cache is full.

EXPERIMENTAL RESULTS

- The authors have tested the storage management and caching capabilities of the system.
- Two workloads used:
  - Set of web proxy logs with 4M entries containing ∼1.86M unique URLs, totaling 18.7GB.
  - File data of ∼2M files from several filesystems, totaling 166.6GB.
- Parameters:
  - Replication factor $k = 5$
  - $b = 4$ (nodeIDs and fileIDs have base 2^b)
  - 2250 PAST nodes

EVALUATING INSERTS AND UTILIZATION

These tests measure the fraction of successful inserts and system utilization. Nodes with several normal distributions of storage capacities were used.

Dist. name	m	σ	Lower bound	Upper bound	Total capacity
d_1	27	10.8	2	51	61,009
d_2	27	9.6	4	49	61,154
d_3	27	54.0	6	48	61,493
d_4	27	54.0	1	53	59,595

Table 1: Distributions of node storage capacities in MB.

STORAGE TESTS

In the first test, replica and file diversion were disabled. 51.1% inserts failed and storage utilization was 60.8%.

The second test compares results with leaf sets of sizes 16 and 32 with $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Dist. Name	Succeed	Fail	File diversion	Replica diversion	Util.
$l = 16$					
d_1	97.6%	2.4%	8.4%	14.8%	94.9%
d_2	97.8%	2.2%	8.0%	13.7%	94.8%
d_3	96.9%	3.1%	8.2%	17.7%	94.0%
d_4	94.5%	5.5%	10.2%	22.2%	94.1%
$l = 32$					
d_1	99.3%	0.7%	3.5%	16.1%	98.2%
d_2	99.4%	0.6%	3.3%	15.0%	98.1%
d_3	99.4%	0.6%	3.1%	18.5%	98.1%
d_4	97.9%	2.1%	4.1%	23.3%	99.3%

A larger leaf set has more successful inserts

A larger leaf set has better utilization but with higher churn overhead

Table 2: Results with different storage distributions and leaf sets.

VARYING DIVERSION PARAMETERS t_{pri} AND t_{div}

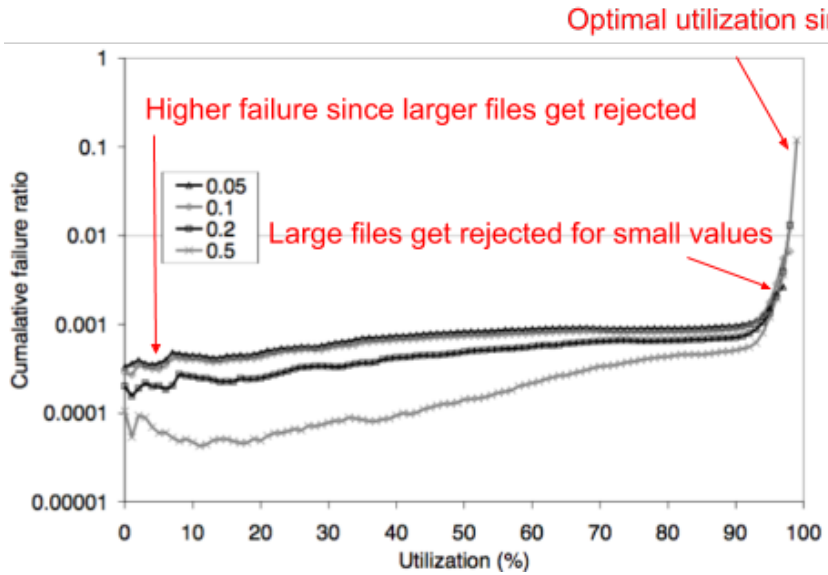

Figure 7: Varying t_{pri} with $t_{div} = 0.05$.

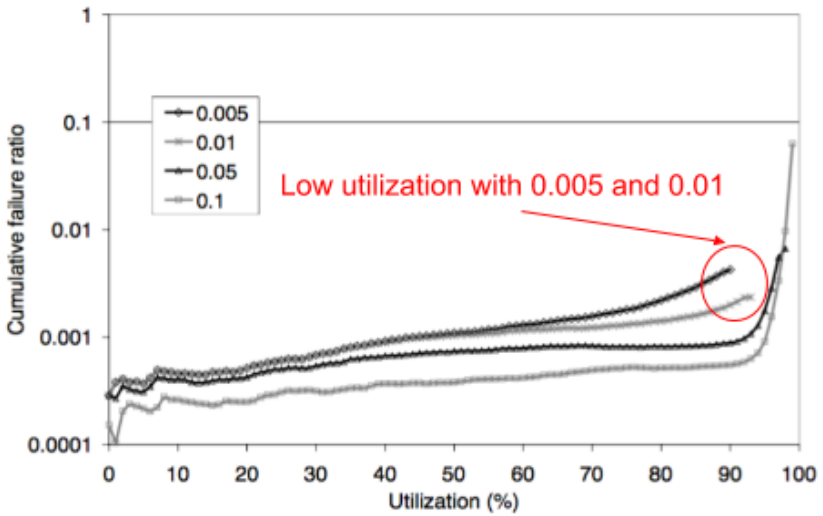

Figure 8: Varying t_{div} with $t_{pri} = 0.1$.

$t_{pri} = 0.1$ and $t_{div} = 0.05$ provide a good tradeoff between utilization and low insert failures.

CACHING

This test compares the GreedyDual-Size (GD-S) algorithm with Least-Recently-Used (LRU).

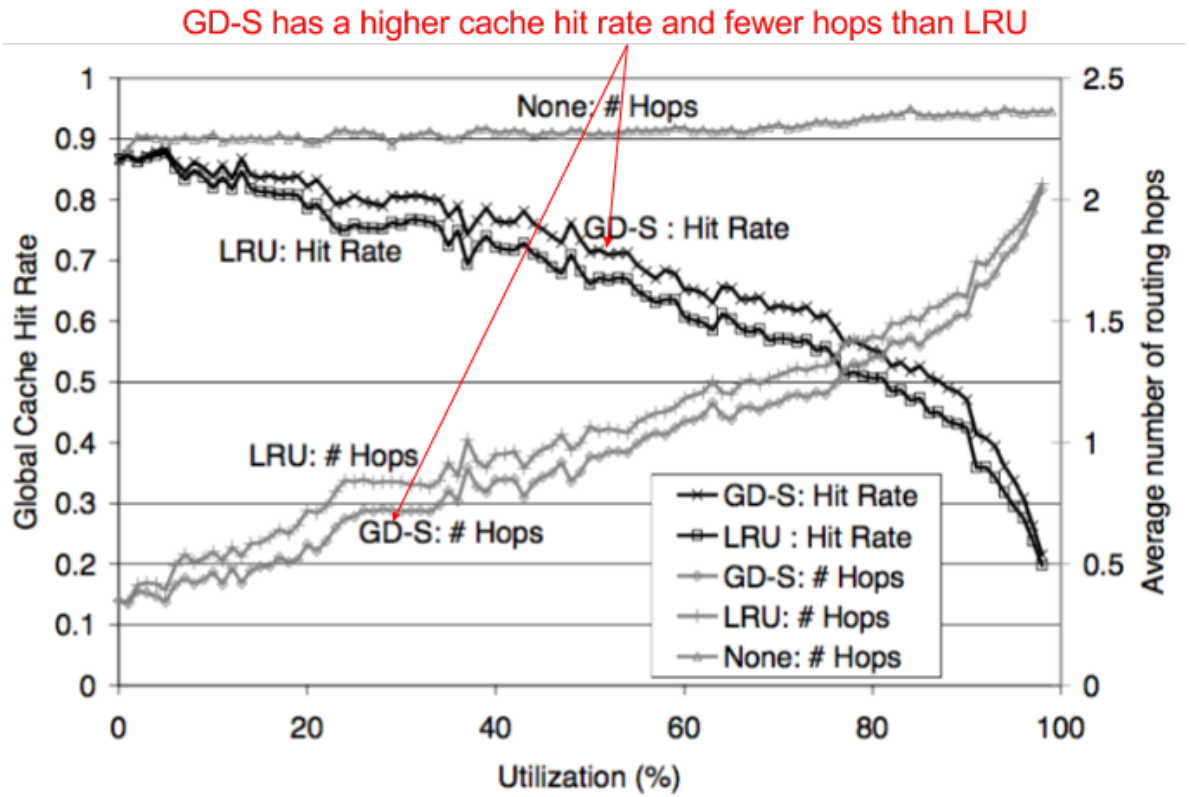

Figure 9: Cache hit rate and routing hops with GD-S, LRU and no caching.

OBSERVATIONS

- Replica and file diversion improve utilization by up to 34-38%.
- A larger leaf set improves file insertion rates and utilization but increases churn overhead.
- Caching reduces query hops even when the system is near full utilization.
- Unless utilization is very high, insertion failure and diversion overhead are low.

COMPARISON WITH CFS

- CFS (Cooperative File System) is a P2P storage system based on Chord, which uses finger table entries to maintain a node's neighbors.
- While PAST nodes store entire files, CFS nodes store file blocks.
- CFS optimizes churn overhead while PAST optimizes query latency.
- PAST enforces per-user storage quotas, whereas CFS limits a per-IP storage quota to x% of the global capacity.

THOUGHTS

- File encoding (e.g. Reed-Solomon) would have reduced storage overhead for high availability.
- PAST doesn't provide searching or strong deletion semantics. Files are immutable.
- If the global storage decreases under high utilization, storing k replicas of files becomes impossible. PAST uses storage quotas to ensure that demand is less than supply.
- Tests don't provide details of churn in the system.
- Tests don't cover related systems such as OceanStore and CFS.