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

Antony Rowstron and Peter Druschel
SOSP '01

Presented by:
Virajith Jalaparti and Giang Nguyen
March 4th, 2010.

PAST

- Peer-to-Peer storage utility
- *(Tries to)* Guarantee (with high probability)
  - Strong persistence
    - Exploits diversity of nodes
  - High availability
    - Replicates data
  - Scalability
    - State logarithmic in number of nodes
  - Security
    - PKI

PAST

- Routing and content location
  - PASTRY
    - *Is this required?*
- Not a general purpose file system
  - Cannot Search for files – *Interesting problem!!*
  - No directories
  - Need *file_id* (distributed offline)
  - Files are immutable

PASTRY (review)

- Incremental Prefix matching

NodeId 10233102

Leaf set (SMALLER | LARGER)

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

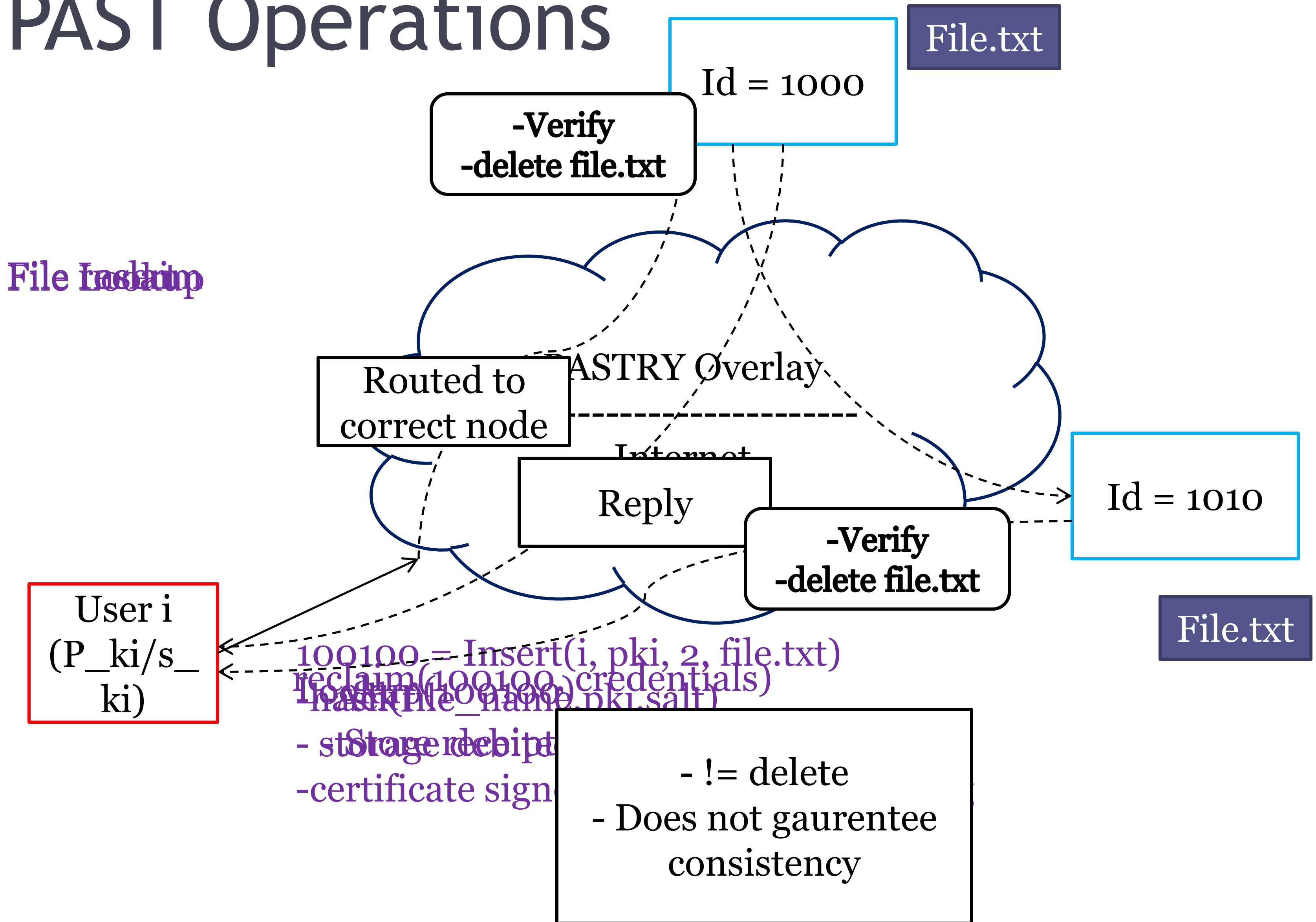
Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

PAST Operations



Storage Management

- Ensure storage utilization as system approaches the maximum
  - Balance free space among nodes
  - K copies of file maintained at nodes closest to file-id
- Reasons
  - Statistical
  - Varying file sizes
  - Varying node storage capacities

An Assumption

- Storage across nodes differs no more than 2 orders of magnitude
- What about large clusters?
  - Multiple PASTRY nodes
  - Should not significantly affect balance
- *Is this correct/justified?*

Replica Diversion

- What happens if a node's free space is reaching 0%?
- Node A cannot hold a file locally
  - File_size > t*free_space_available (tpriv and tdiv)
  - Node B in leaf set
    - Not one of k closest nodeids
    - Does not already have a copy
  - Stores a pointer to B

Replica Diversion

- No such B exists
  - File diversion
    - Create a new id for the file to store in another part of ring
- Handling Failures
  - A fails
    - Another node C (having (k+1)st closest id) points to B
  - B fails?

Maintaining Replicas

- Invariant: K copies at all times
- PASTRY helps detect failures/additions
  - A new node becomes one of the k-closest nodes to a file
- Copy the file to the newly added node
  - Optimization: new node points to old node, if it exists
  - Eventually copied to new node
- Very High storage utilization
  - <k copies of file.

Caching

- Popular files cached (>k copies)
  - A node that participates in routing a file caches it
- Unused space at a node used for caching
- Cached files evicted if new files are to be stored
  - GreedyDual-Size policy used
    - File with lowest (*access_count/size)* ratio evicted
      - *Unfair for large files*

PAST Security

- Public/Private key-pair
  - Smart cards
  - *PKI*
- Owner file/reclaim certificate
  - Prevents masquerading
- Store/Reclaim receipts
  - Ensures k copies are stored/reclaimed
- Randomized routing in Pastry to prevent continuous interception
  - *How would this affect bounds guaranteed?*

Eval (1)

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: The parameters of four normal distri tions of node storage sizes used in the experime All figures in MBytes.

With no diversions and using distribution d1: 51% insertions fail, global utilization 60%

Web proxy log trace: 1.86 million objects, totaling 18.7 GB

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%

Table 2: Effects of varying the storage distribution and leaf set size, when $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Eval (2)

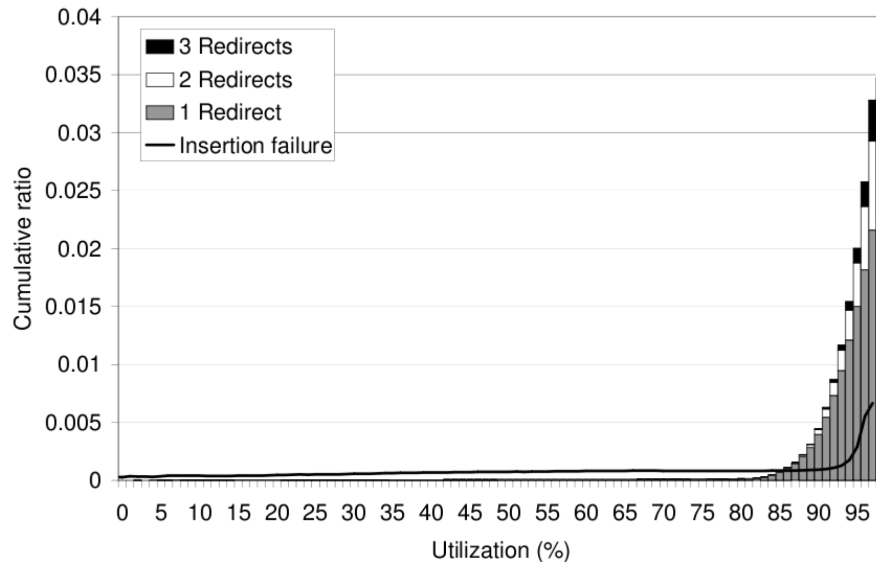


Figure 4: Ratio of file diversions and cumulative insertion failures versus storage utilization, $t_{pri} = 0.1$ and $t_{div} = 0.05$.

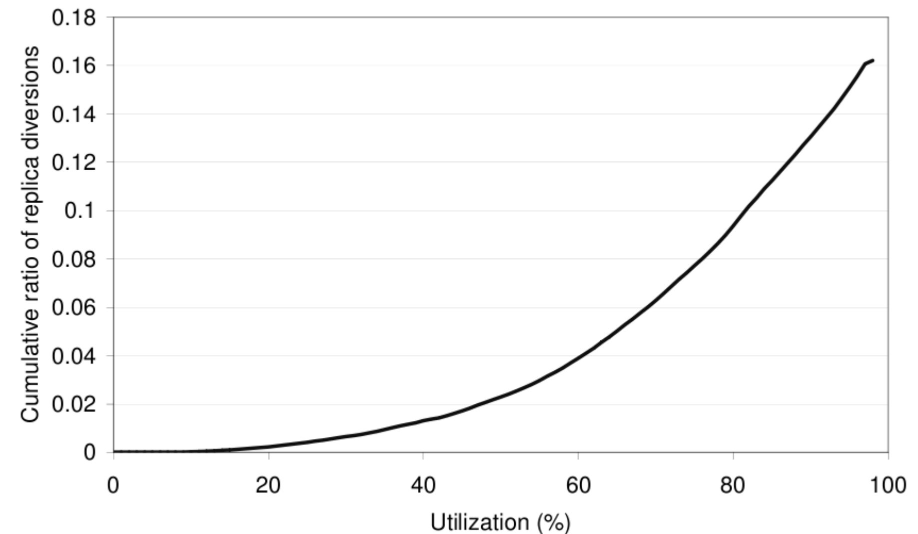


Figure 5: Cumulative ratio of replica diversions versus storage utilization, when $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Eval (3)

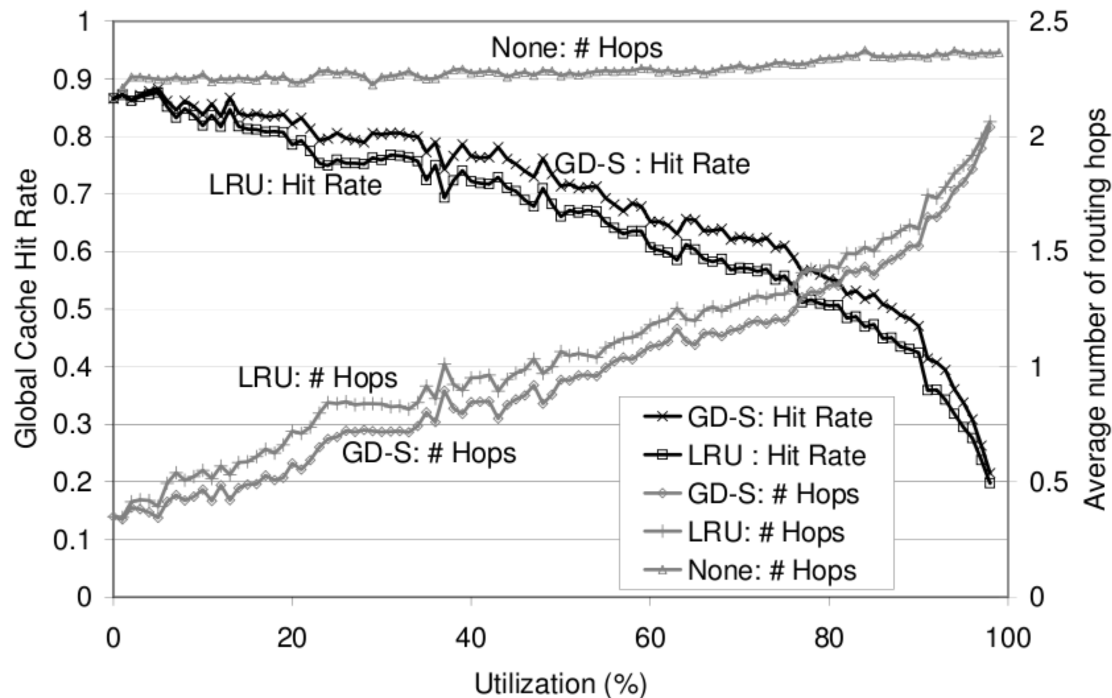


Figure 8: Global cache hit ratio and average number of message hops versus utilization using Least-Recently-Used (LRU), GreedyDual-Size (GD-S), and no caching, with $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Discussion

- Can I always get back my file?
- Churn
- Storing of large files
  - CFS is block oriented
  - Tradeoff?
- Why would any one use P2P storage?
  - Economics
  - User Quotas!
  - Oceanstore
    - Uses Tapestry
    - Money involved
    - Hierarchical

UsenetDHT: A low-overhead design for Usenet

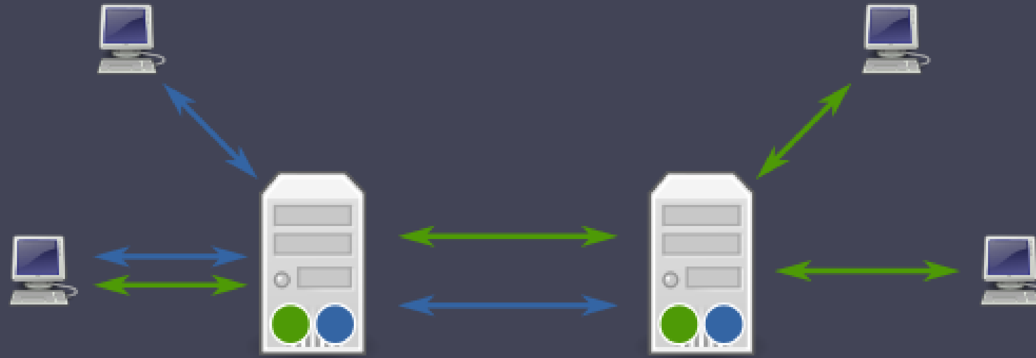
Emil Sit, Robert Morris, and M. Frans Kaashoek

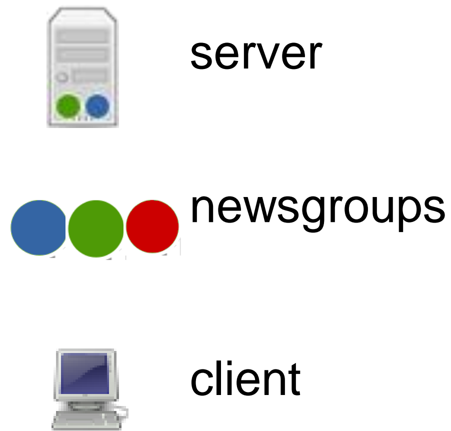
Presented by Giang Nguyen
March 04, 2010

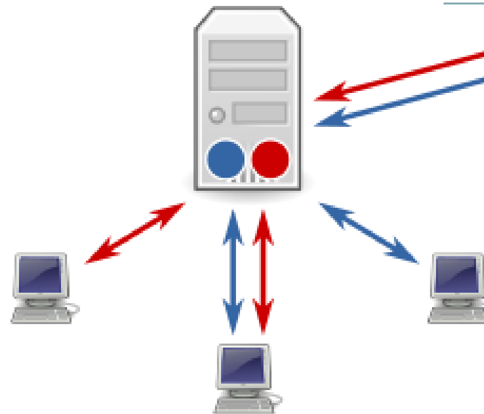
Background

- Usenet: a news system operating since '81
- Similar to online forums/bulletin boards
- Overview
  - Newsgroups such as “class.sp10.cs525” or “comp.lang.python”
  - A Usenet server stores articles for newsgroups of its choosing
  - Paying users post/download articles to/from their “local” (e.g., ISP) Usenet server
  - Usenet servers flood articles to interested peers
    Retention policy: how long a server retains articles

Background







Source: Benjamin D. Esham/Wikipedia

Motivation

- Usenet still popular:
  - Sept. '07, usenetserver.com saw 40,000 concurrent clients, 20 Gb/s
- Growing content volume, largely binary articles
- Top servers receive from peers 3 TB/day
- Each server stores articles for its interested newsgroups --> lots of duplication

Chord ring review

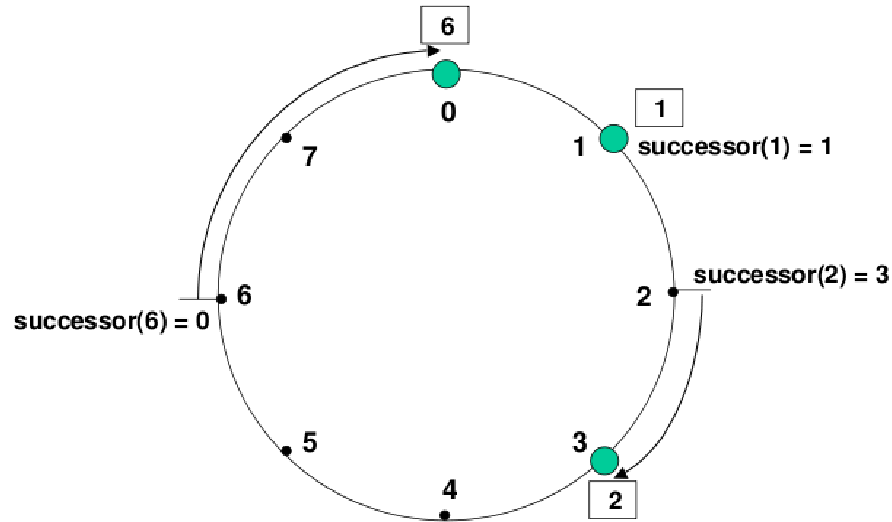


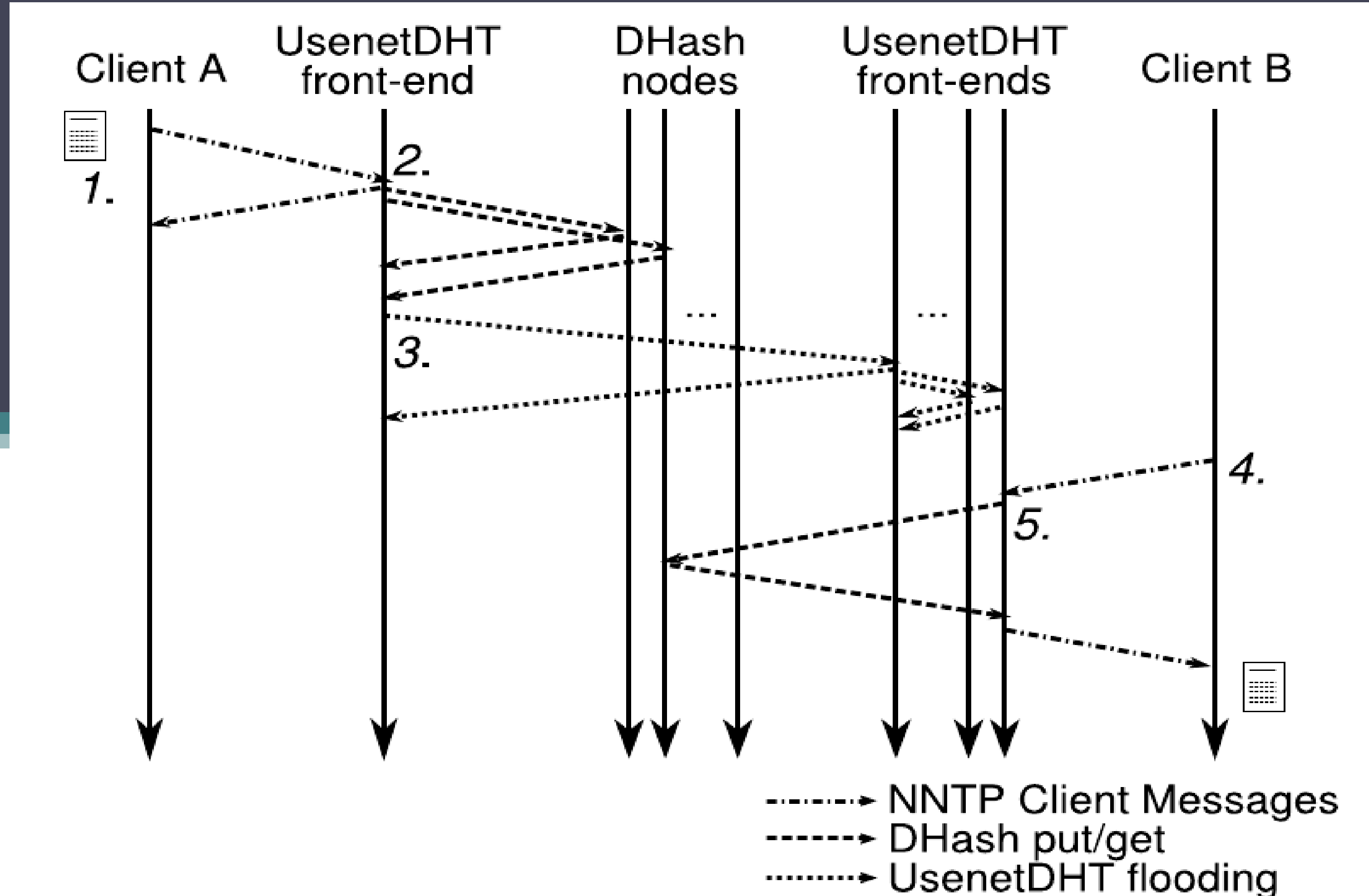
Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Source: I. Stoica et al. *Chord: a scalable peer-to-peer lookup service for Internet applications*

UsenetDHT

- Mutually trusting peer sites use (DHash) DHT to store articles (content but not metadata)
- "Front-ends" preserve the client- and external peer-facing interface (NNTP protocol)
  - Translate clients' post/download to `put`/`get` requests to DHT storage
  - Key == SHA-1 of article's content
  - Cache article content to reduce load on DHT storage
  - Flood article's metadata to peer front-ends
  - Store all articles' metadata

UsenetDHT



UsenetDHT replication

- Goal: store replicas at *k* (= 2) successor nodes
- Requires keeping track of current number of replica
- Challenges:
  - no efficient data structure that supports both insertion/lookup of millions of entries and synchronization among nodes
  - high write rates --> local state quickly becomes stale
- Solution: Passing Tone algorithm:
  - local: each node ensures it has replicas of articles for which it is responsible
  - global: it ensures articles for which it no longer is responsible are given to new responsible node

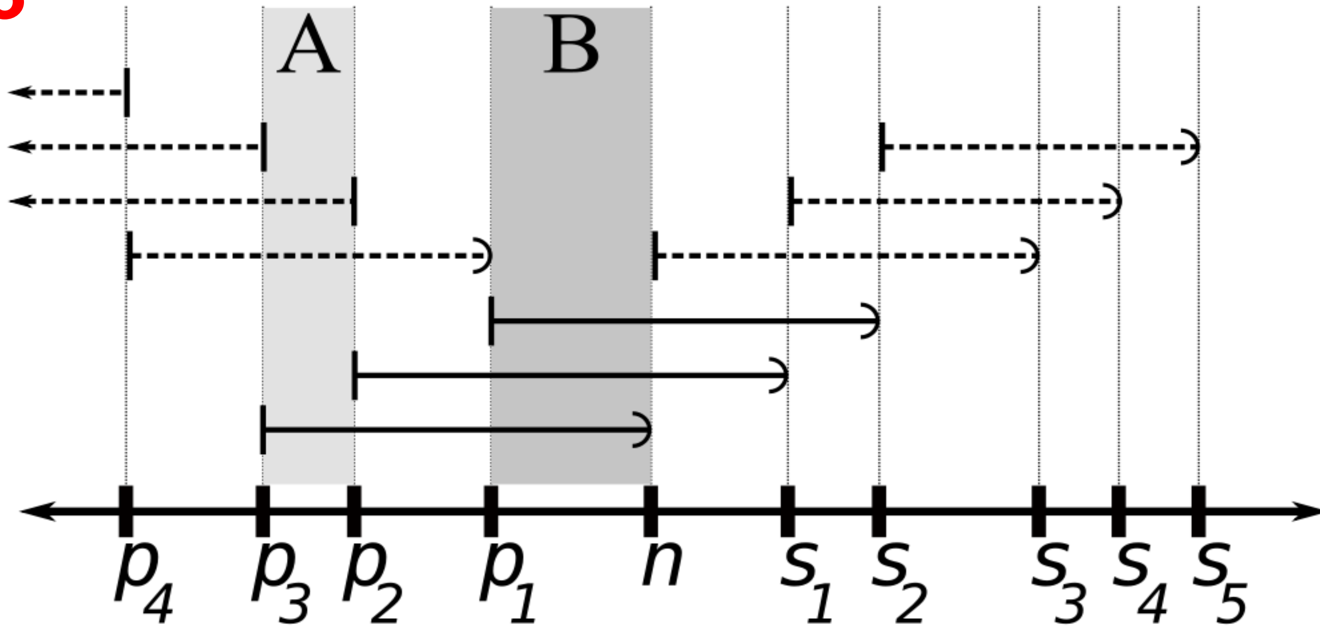
Local maintenance

- Each node ensures it has replicas of articles for which it is responsible
- Each node has a hash/Merkle tree of the keys that it has
- Replicates articles from immediate predecessor & successor
- Algorithm @ node *n*:

```
a, b = n.pred_k, n   # "a" is k-th predecessor
for partner in n.succ, n.pred:
    diffs = partner.synchronize(a, b)
    for o in diffs:
        data = partner.fetch(o)
        n.db.insert(o, data)
```

Local maintenance

k = 3



- When n fails, s1 becomes responsible for storing replicas of articles in region A and will obtain them (most likely) from p1
- When n joins/returns from failure, it obtains articles in region B from s1

Global maintenance

- Each node ensures articles for which it no longer is responsible are given to new responsible node
- Algorithm @ node *n*:

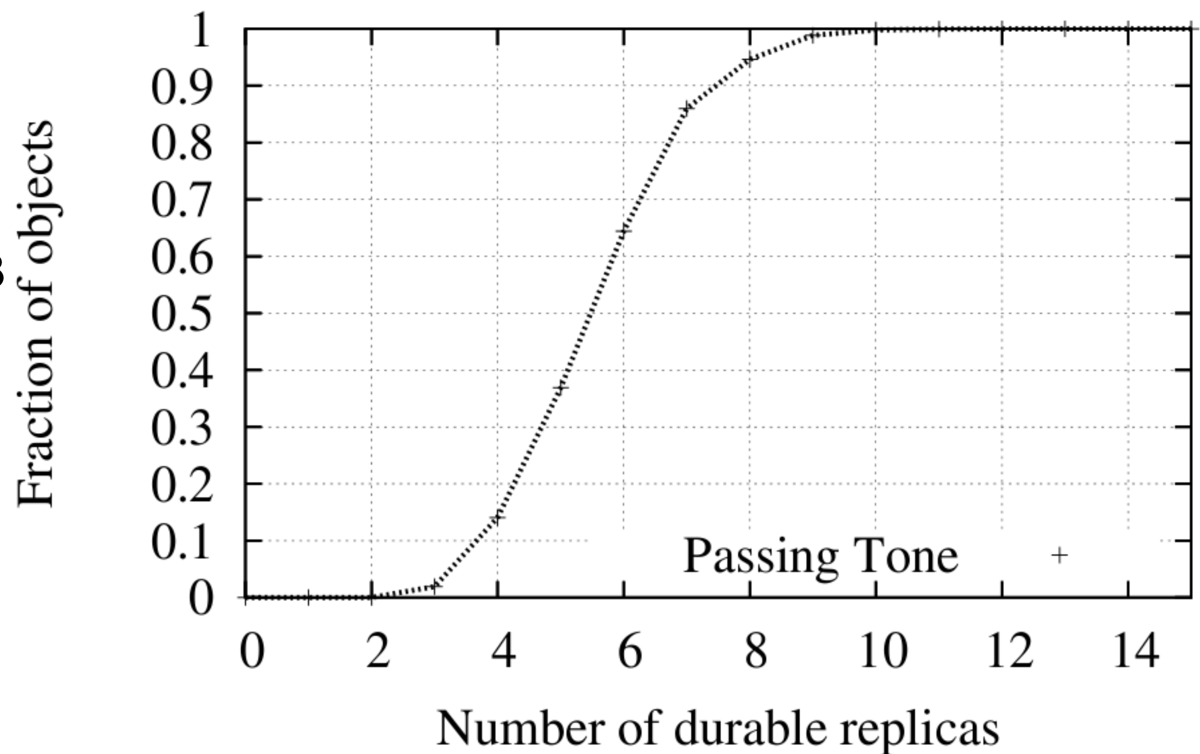
```
a, b = n.pred_k, n   # "a" is k-th predecessor
key = n.db.first_succ (b) # first key after b
while not between (a, b, key):
    s = n.find_successor (key)
    diffs = s.reverse_sync (key, s)
    for o in diffs:
        data = n.db.read (o)
        s.store (o, data)
    key = n.db.first_succ (s)
```

Supporting article expiration

- Usenet articles have a retention period
- The maintenance algorithms oblivious of article expiration --> might replicate expired articles --> wasteful
- One solution: add metadata to the Merkle sync tree --> complicates construction & requires custom tree per peer (why?)
- Approach taken: while constructing the tree, skip expired articles
  - --> requires nodes to have sync'ed clocks
  - What if you don't/can't have sync'ed clocks?

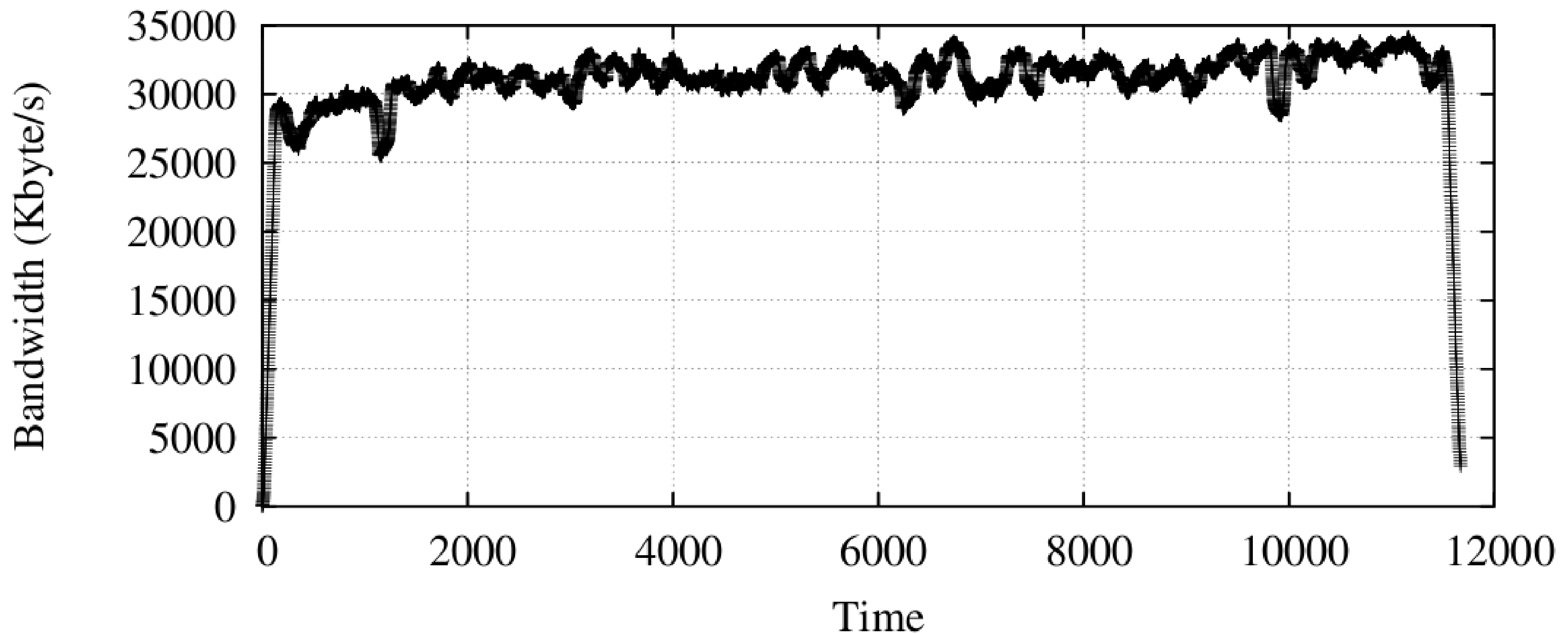
Simulation: article durability

- 632-host Planetlab trace Mar. '05 to Mar. '06
- 21,255 transient failures & 219 disk failures
- Pre-populated with 50K 20KB articles, 150 KB/s for maintenance
- k = 2



- All articles had at least 3 replicas
- No article was lost

Live deployment



12 geographically dispersed but well-connected nodes across US
Load: mirror of CSAIL feed (2.5MB/s writes)
Aggregate read throughput test: 100 PlanetLab clients, each running 5 parallel NNTP clients, continuously fetching articles from random newsgroups
-> sustained aggregate throughput: ~30MB/s

Criticisms

1. Claims bandwidth savings but no evaluation
   - Front-end read cache not implemented
2. All participating cites have to use same policies for filtering, retention, etc
3. Requires synchronized clocks to prevent spurious repairs
4. No discussion on article locality
5. Motivation: numbers from ("top") commercial Usenet providers. Solution: target mutually trusting universities
   - MIT CSAIL's feed is 619th in top 1000 Usenet servers

Discussion questions

1. What if we use hash of article's header as Chord key?
2. Could we exploit article locality (a thread)?
   1. A thread as the DHT unit? Front-ends store/retrieve the whole thread, cache, and return the requested article?
3. What if we use finer granularity of content blocks?

Possible answers

1. Less space-efficient if different articles (headers) have same content, because they'll get different keys. (Analogous to not supporting hard links.) Is there a DHT that supports different keys pointing to the same storage?

CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups

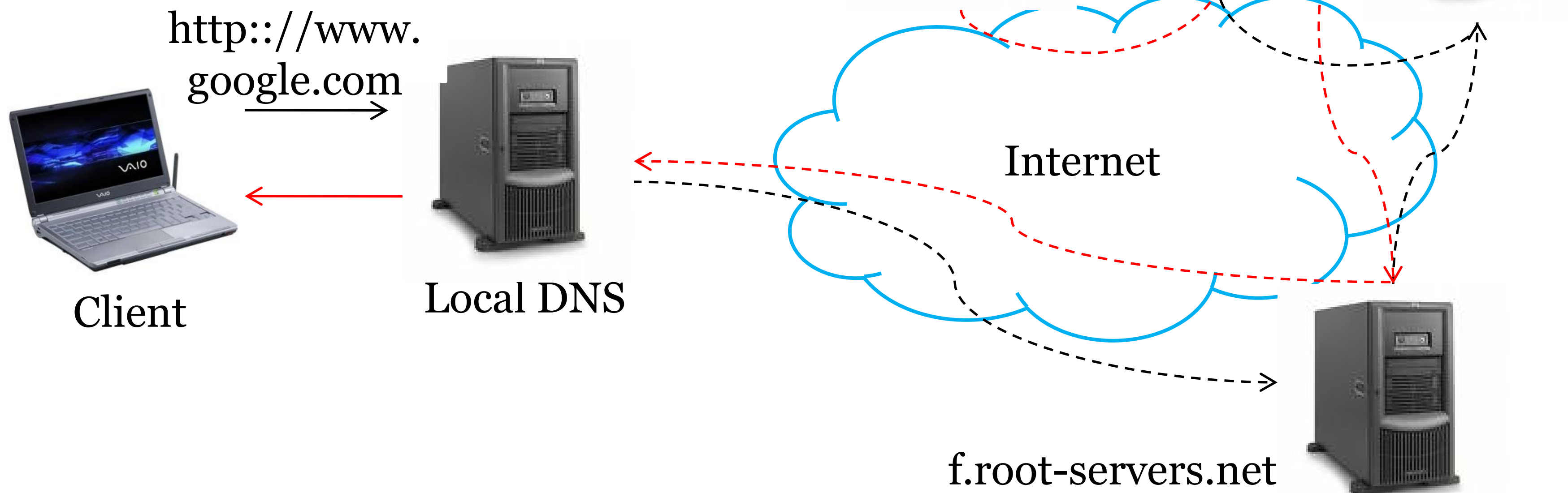
KyoungSoo Park, Vivek S. Pai, Larry Peterson, and Zhe Wang

OSDI '04

Presented by:
Virajith Jalaparti
March 4th, 2010.

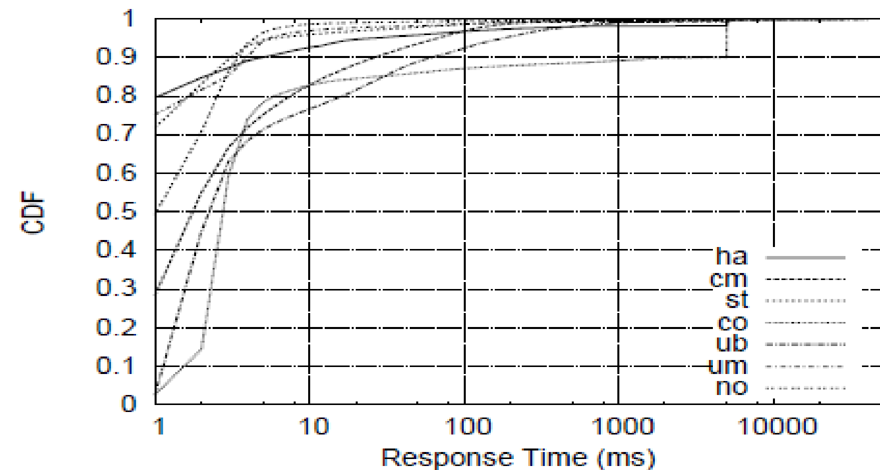
Domain Name System

- Distributed lookup service
  - Hostname → IP address
- Widely used



Problems with the DNS

- Server Side
  - Scalability
  - Reliability
  - Mis-configurations
  - *Censorship*
- Client Side
  - 90% hit rates; aggressive caching, redundancy
  - Several problems still exist: dealt with in this paper - occasional large response times



(a) Percentage of lookups taking < x ms

Reasons

- Packet Loss
  - 0.09% for 4 hops
  - *Should also consider overloading of swicthes near LDNS*
- Resource Competition
  - DNS is supposed to be light weight
- Maintenance problems
- LDNS overload

Cooperative DNS

- Goals
  - Incrementally Deployable
  - Minimal extra resource usage
- Acts as intermediate between client and LDNS
  - Deployed on PlanetLab
  - Queries LDNS on behalf of client
  - Remote DNS servers used if LDNS doesn't respond quickly
- Insurance Model used

Design Considerations

- Proximity and availability of a remote LDNS
  - Heart-beats
  - *Currently not very adaptive*
- Locality considerations
  - Highest Random Weight hashing scheme
- Which remote server to choose?
  - Total time should be less than local lookup time.

Design Considerations

- How may simultaneous look-ups needed?

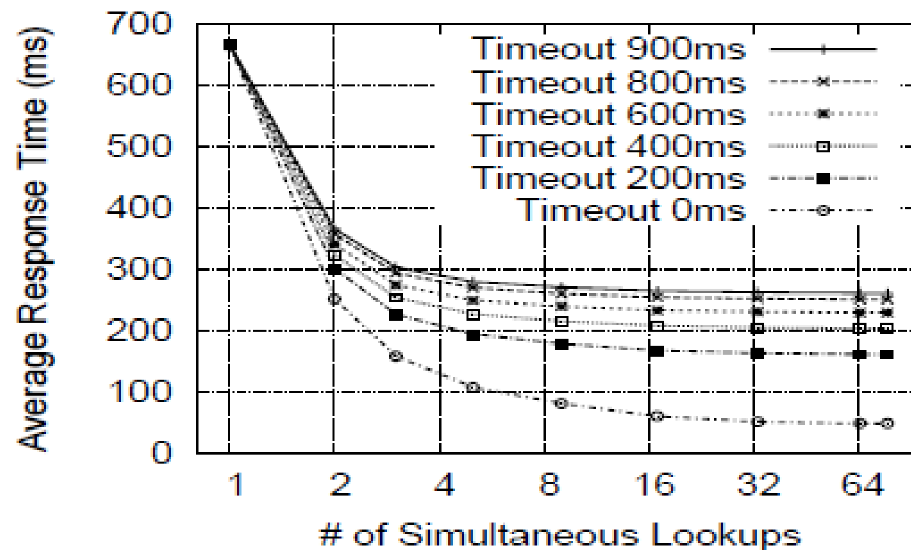
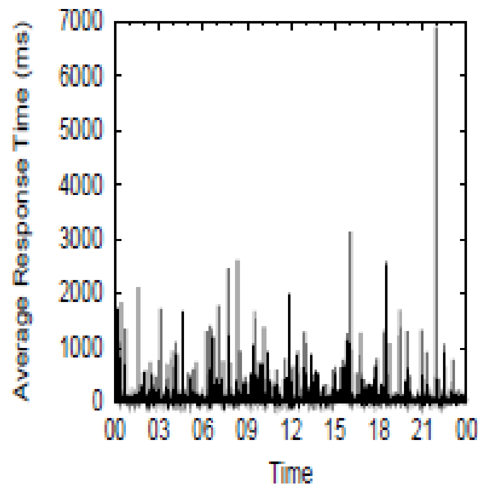


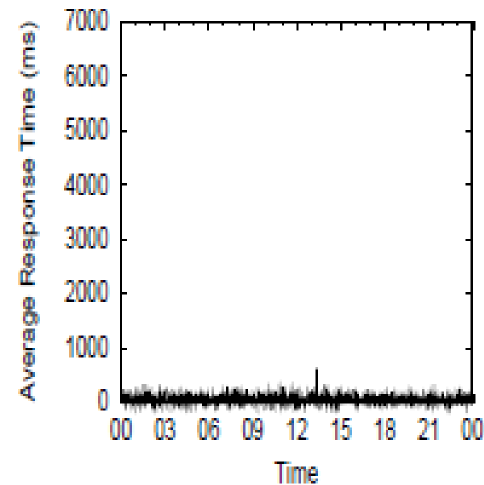
Figure 10: Average Response Time

Evaluation – Response time

- 1day Live Traffic from CoDeen



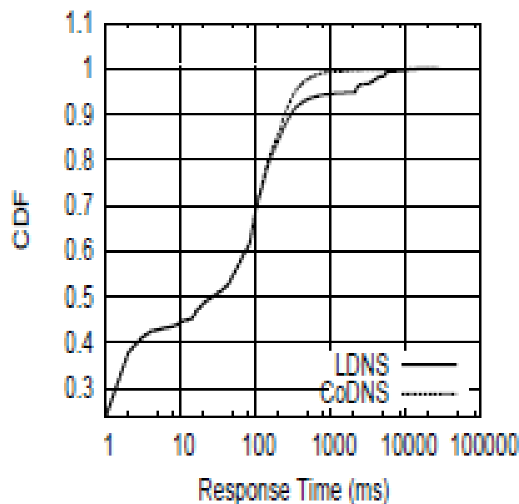
(a) Local DNS



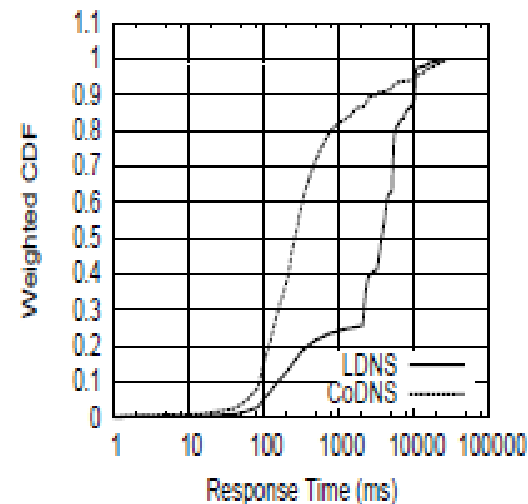
(b) CoDNS

Evaluation – Response time

- 1day Live Traffic from CoDeen



(a) Response Time CDF



(b) Total Time CDF

Is CoDNS perfect?

- Large number of unsuccessful remote queries

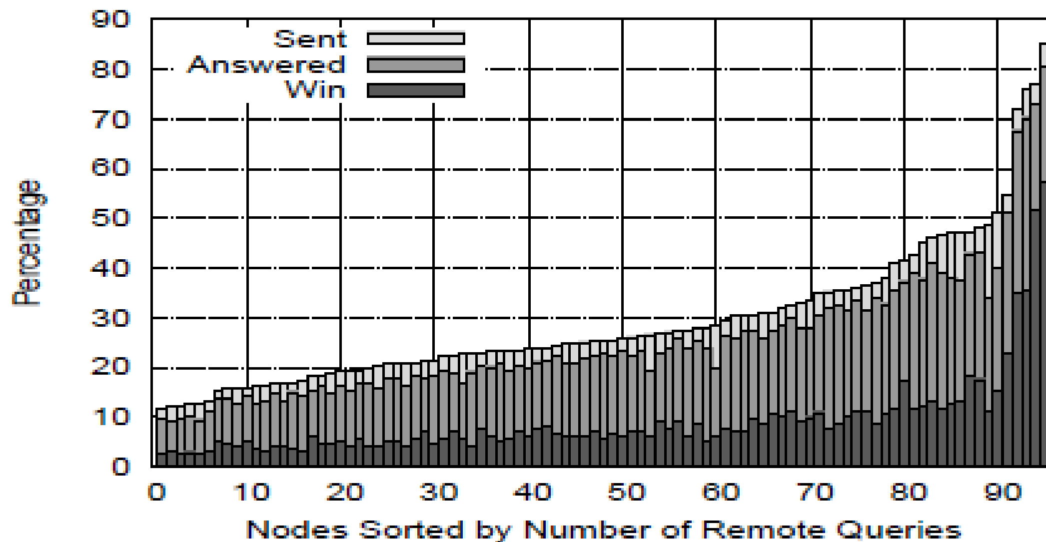


Figure 19: Analysis for Remote Lookups

Is CoDNS perfect?

- Large number of unsuccessful remote queries
- *Improvements*
  - *Better prediction mechanisms*
    - *Leverage ML*
  - *Choose better remote sites*

Discussion

- Security Issues of CoDNS
  - Has more vulnerabilities than DNS
  - DNSSEC
  - Major problem in existing widely deployed applications
- Correlated failures
- Other methods to remove client-side problems
  - Use multiple LDNSs
    - balance load among them!
    - Anycast
    - Valiant Load balancing
    - $$!!
- DNS and DHTs??