CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups

KyoungSoo Park, Vivek Pai, Larry Peterson and Zhe Wang

Presented by Raoul Rivas and Chihung Lu

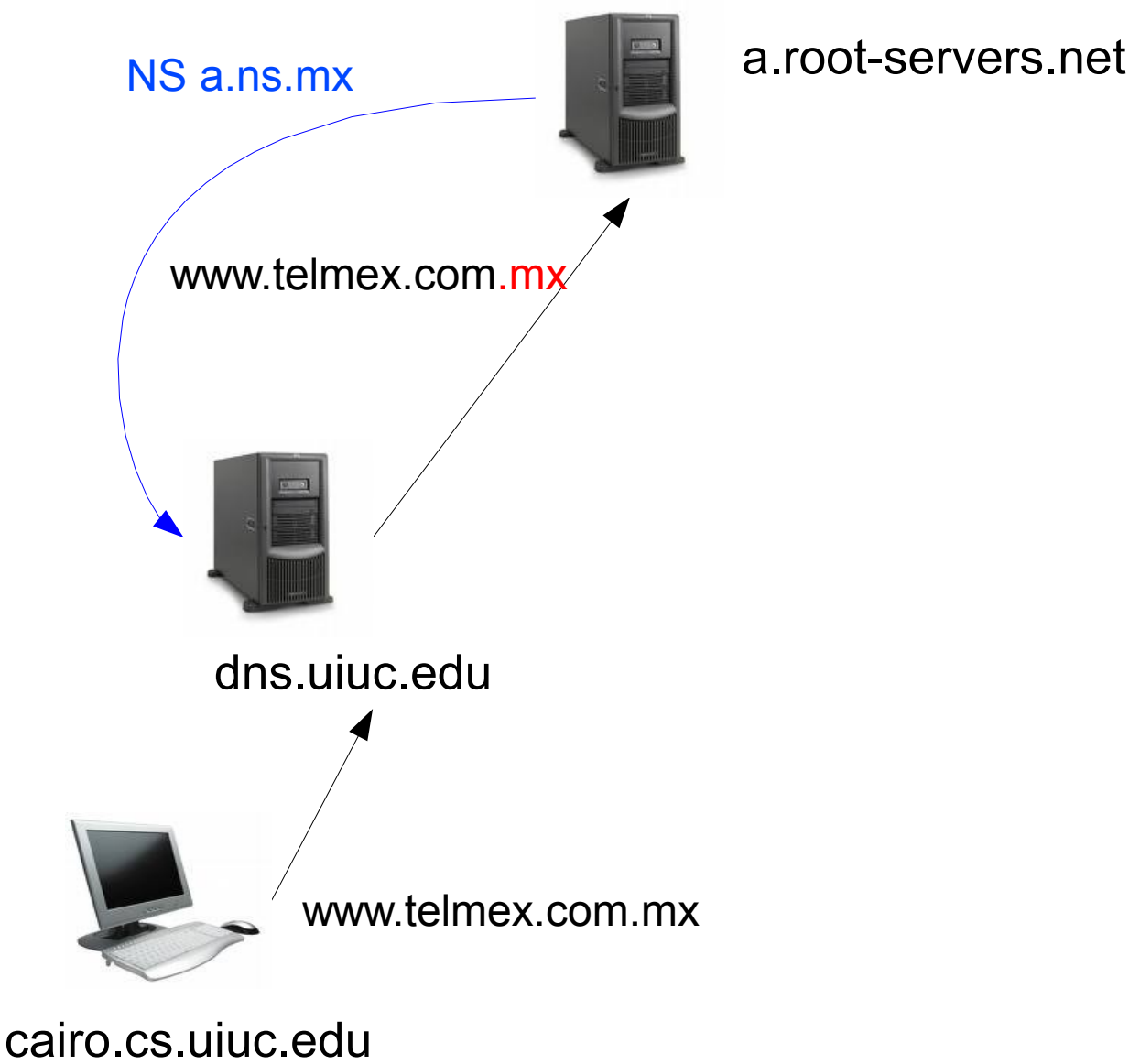
DNS Overview

dns.uiuc.edu

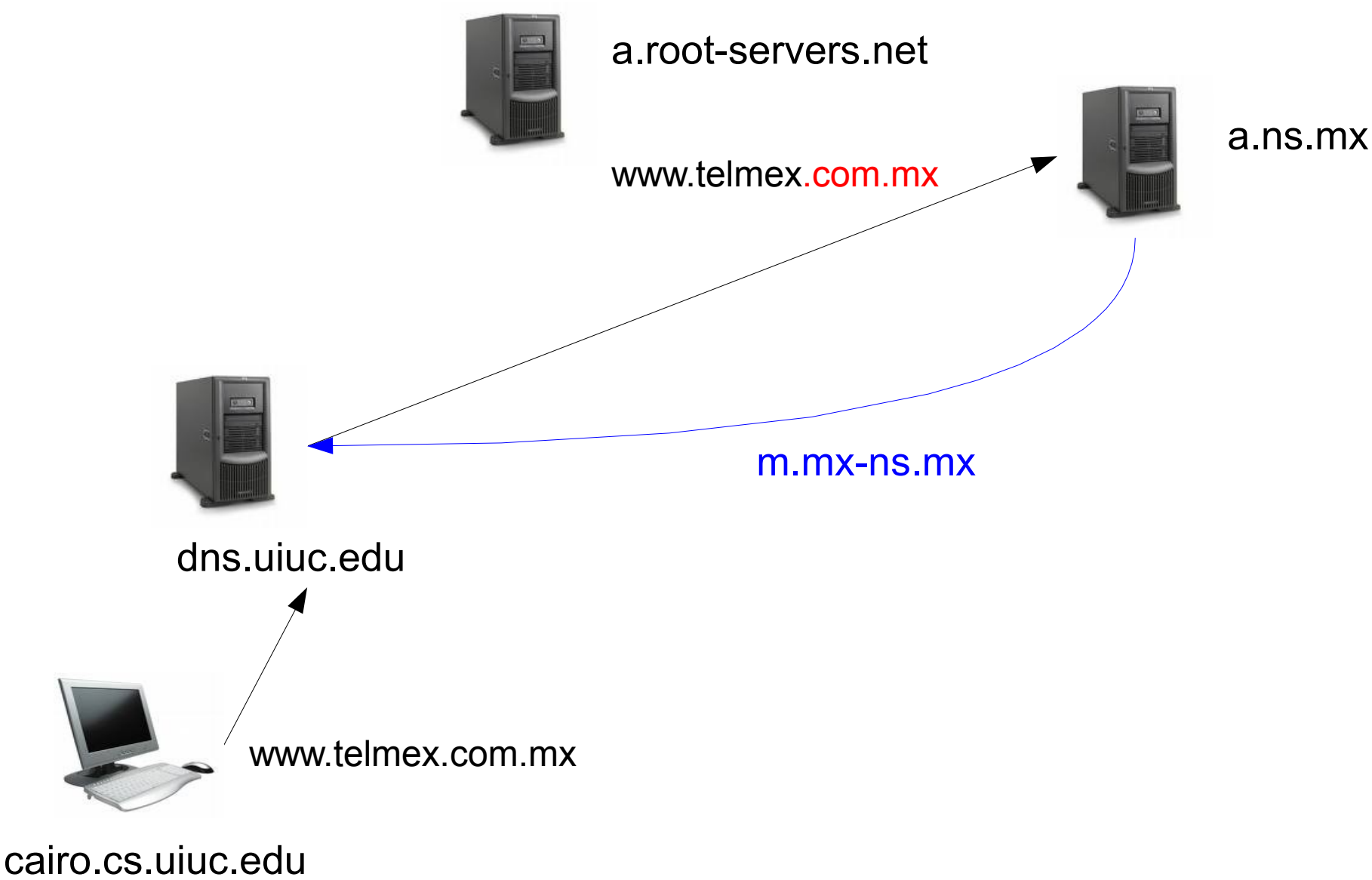
www.telmex.com.mx

cairo.cs.uiuc.edu

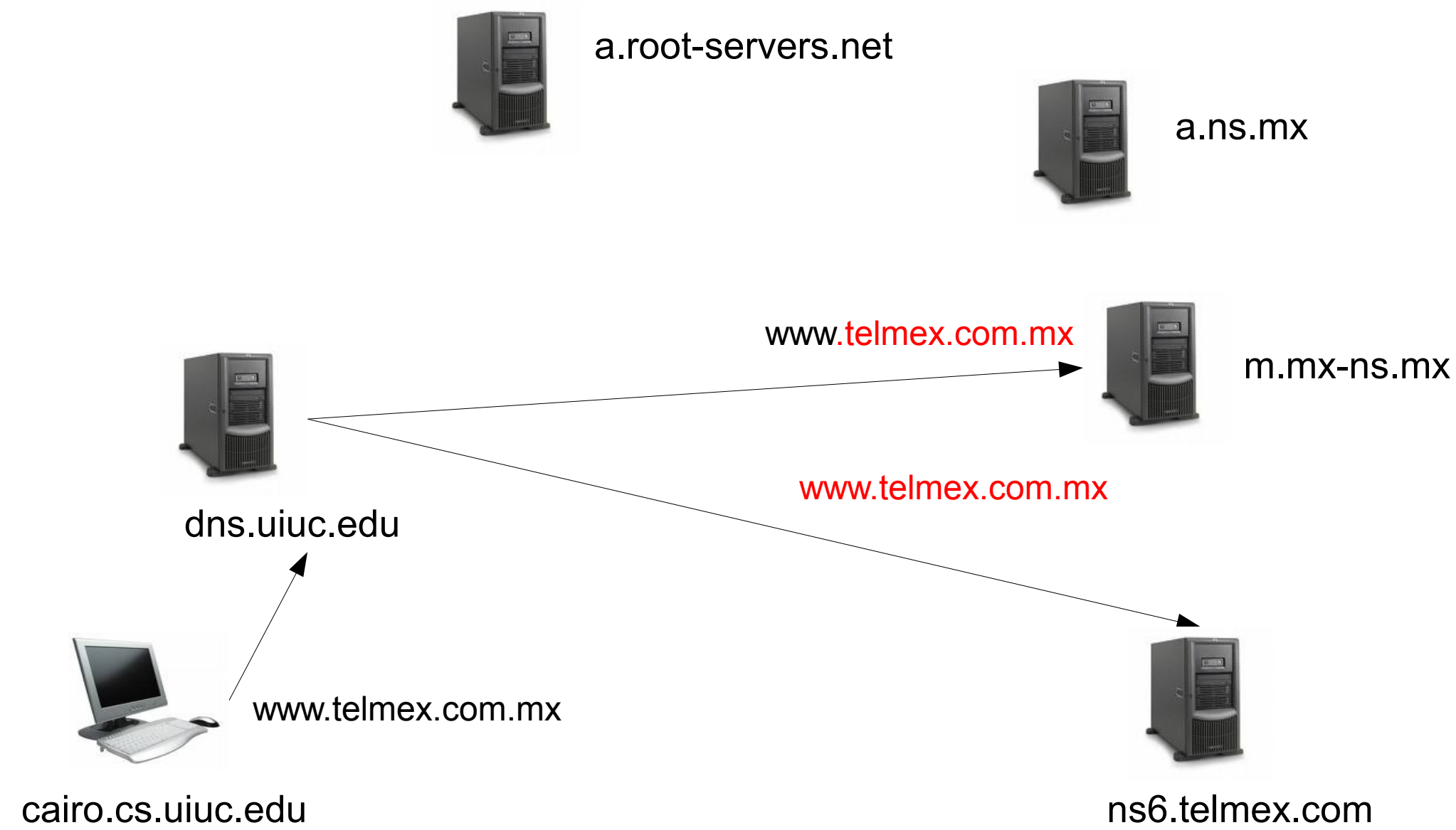
DNS Overview



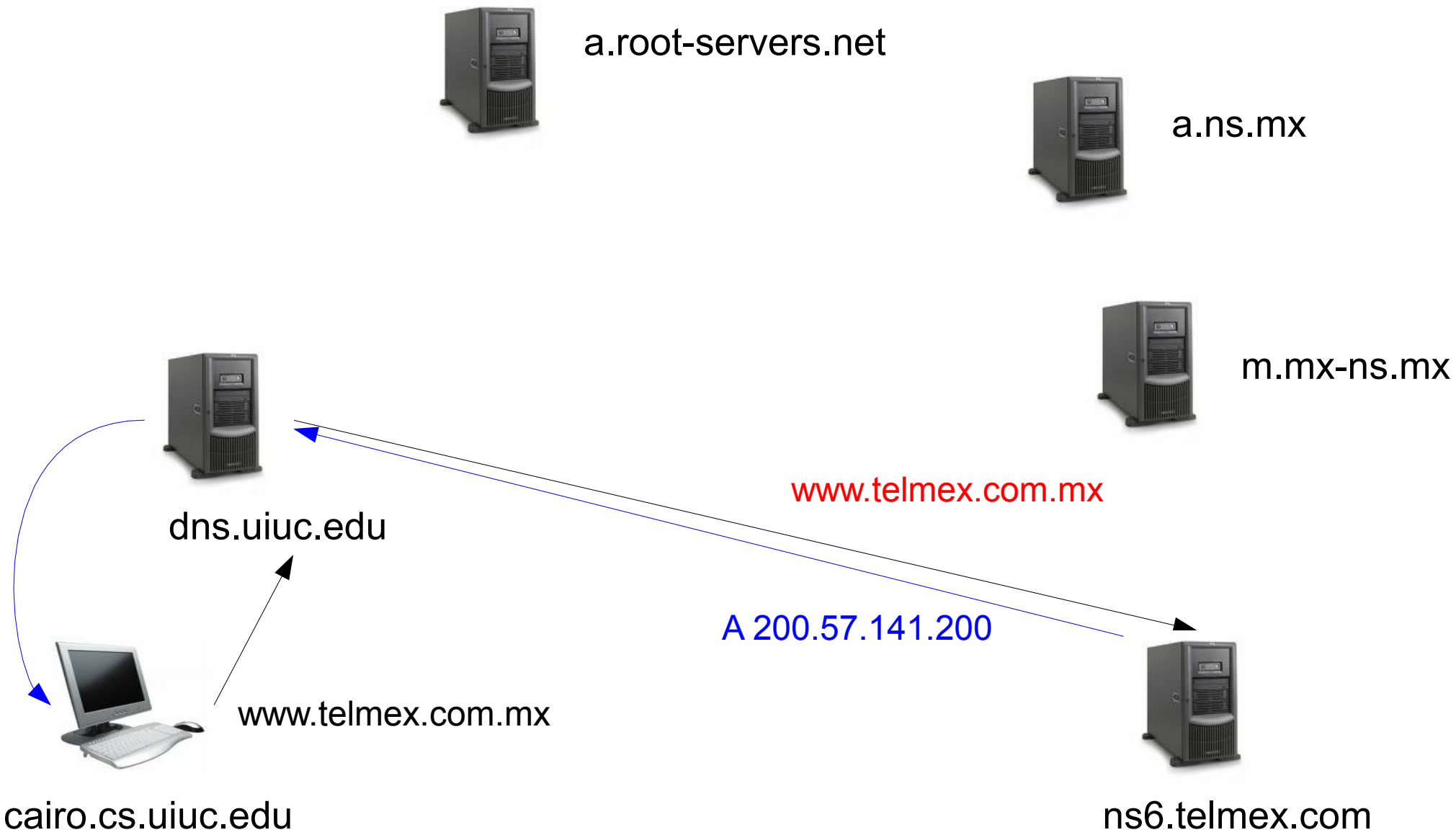
DNS Overview



DNS Overview



DNS Overview

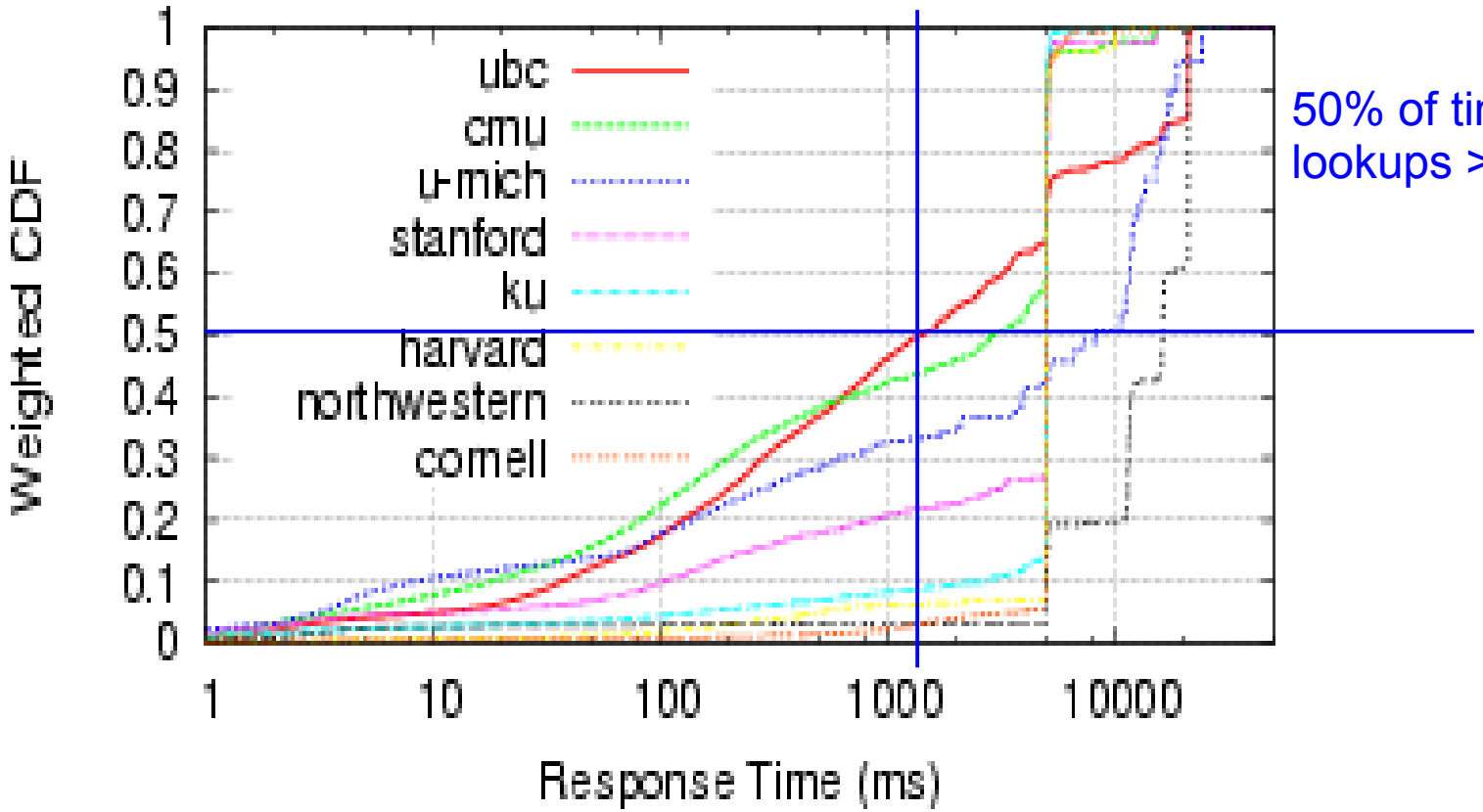


Motivation and Significance

- DNS failures can be divided in client-side or server-side
- Client Side failures are frequent and degrade performance and reliability of DNS
  - Most DNS lookup response times are between 10 ms and 1000 ms but they can be as large as 100,000 ms
- DNS lookups contribute more than 1 second in 20% of the retrievals of web objects [CE Wills *et al.*]

Motivation and Significance

- Most DNS failures are transient
- A small percentage of failure cases dominates the total response time



50% of time spent on lookups >1000 ms

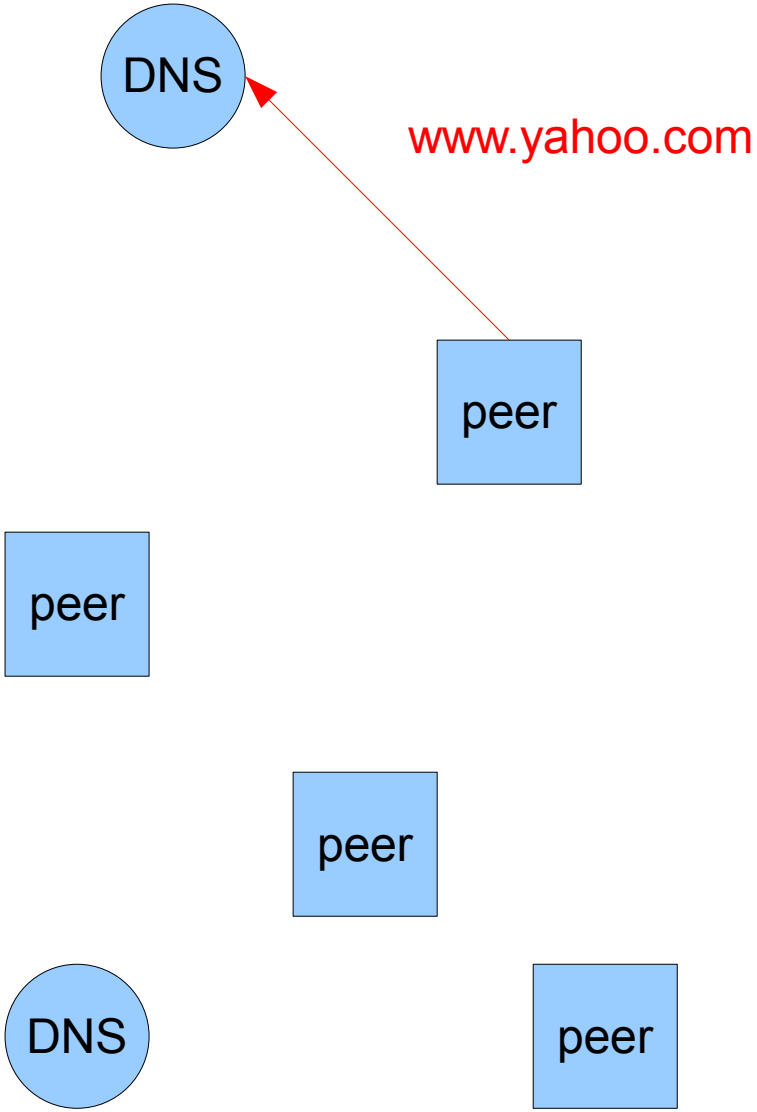
Origin of Client-Side Failures

- Packet Loss
  - DNS servers can be a few hops far away
- Overloading of the nameserver
  - High bursty traffic causes UDP buffer to fill
- Maintenance Mistakes
- Heavy non-related processes running on the DNS server

Design Constraints

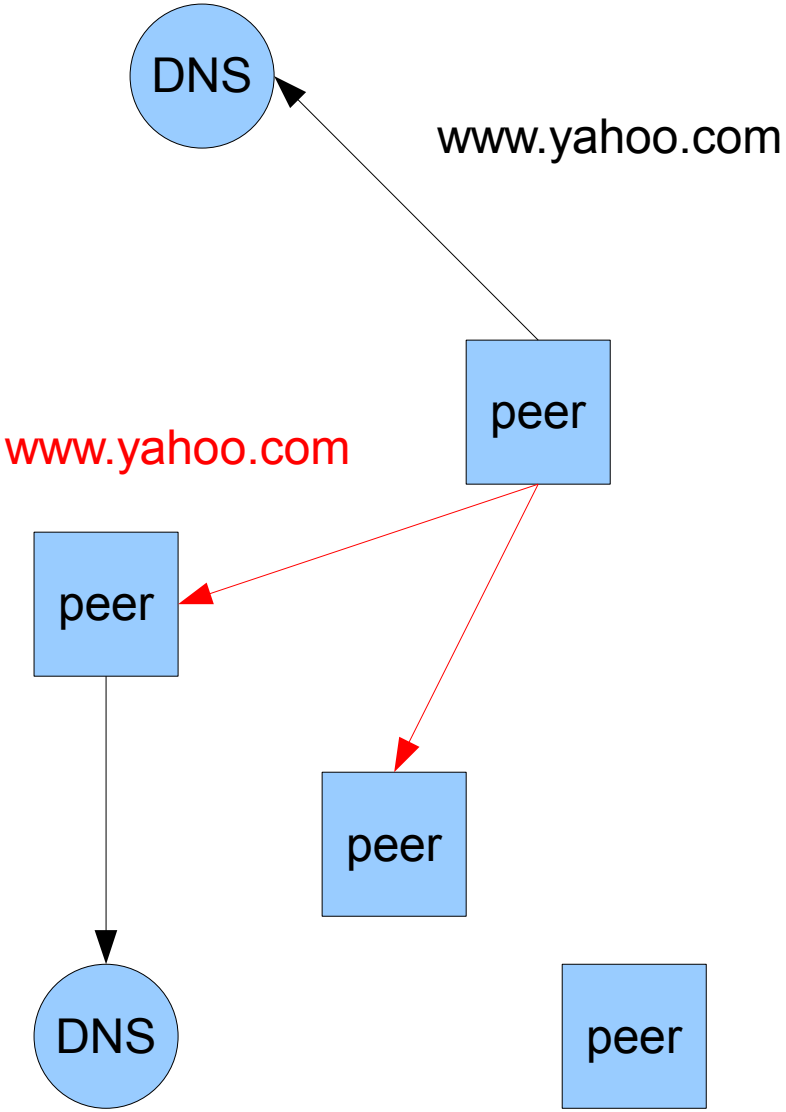
- Incrementally deployable
- Minimal Resource requirements
- Failures on DNS are uncorrelated
- Number of “healthy” nameservers is always high
- DNS requests and responses are small
  - Focus on Low latency rather than high bandwidth

Lookups in CoDNS



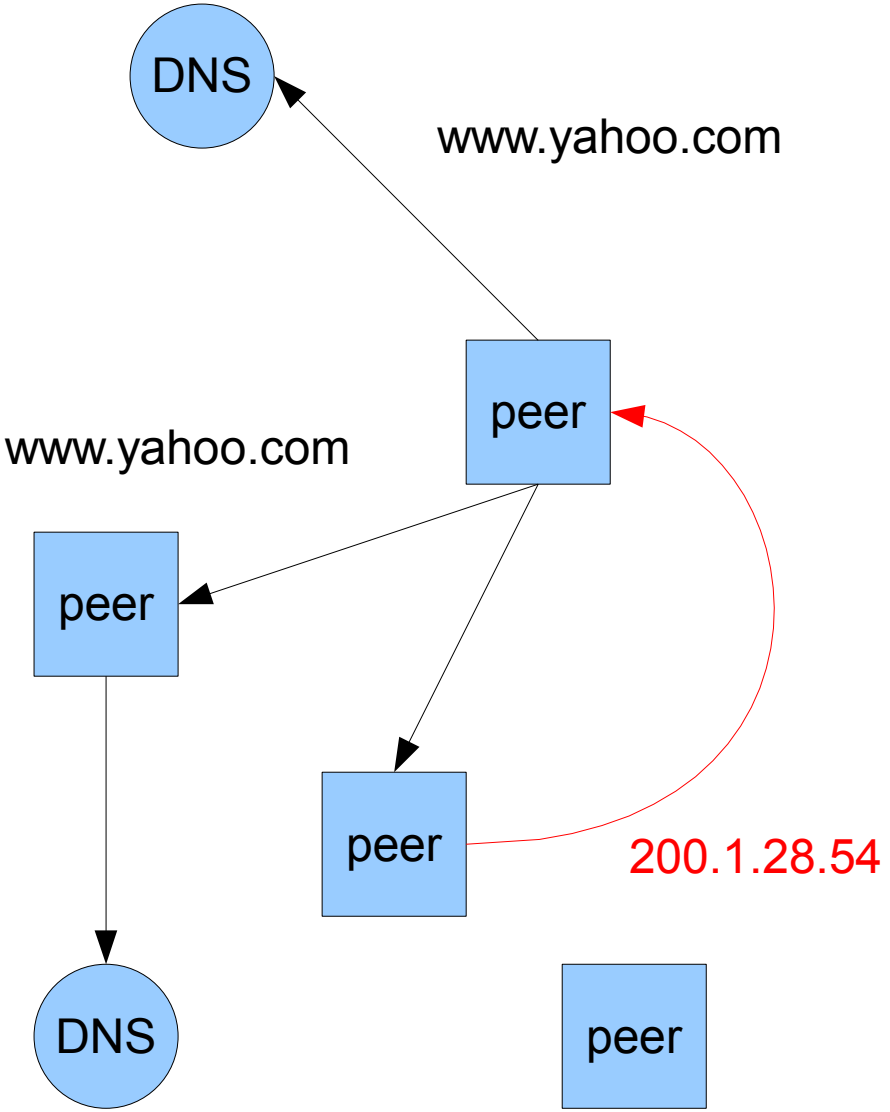
- Contact the local nameserver first
- Wait for a dynamically established timeout (200ms initially)
- Pick n-1 peers from a list of CoDNS peers
- Issue n-1 simultaneous lookups
- Use the first response either from a peer or DNS

Lookups in CoDNS



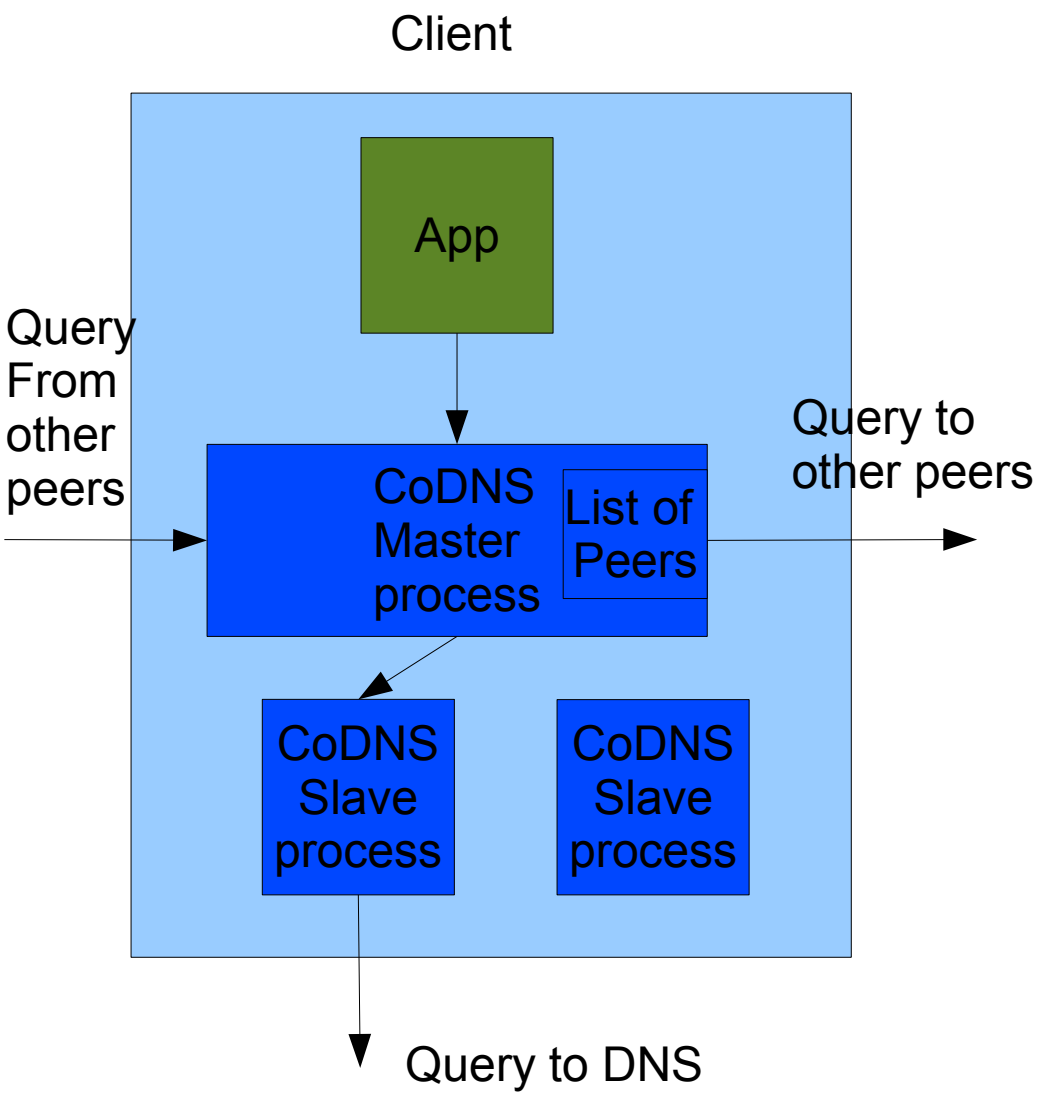
- Contact the local nameserver first
- Wait for a dynamically established timeout (200ms initially)
- Pick n-1 peers from a list of CoDNS peers
- Issue n-1 simultaneous lookups
- Use the first response either from a peer or DNS

Lookups in CoDNS



- Contact the local nameserver first
- Wait for a dynamically established timeout (200ms initially)
- Pick n-1 peers from a list of CoDNS peers
- Issue n-1 simultaneous lookups
- Use the first response either from a peer or DNS

Architecture and Design



- CoDNS is implemented as a daemon in the client
- A non-blocking Master process receives lookup queries from the Application or other Peers
- The Master process forwards the query to a blocking slave process

Architecture and Design

- The Master process is in charge of the following:
  - Record the timeout of the query
  - Send and Receive remote queries
  - Select a peer based on the Highest Random Weight Hash
    - Consistent Hashing, low overhead, minimal impact if a server leaves or joins, load balanced [Thaler *et al*]
  - Monitor peers health by sending a periodic heartbeat

Architecture and Design

- Peer queries are sent only after trying to obtain a fast response form the DNS
  - Lookups are generally fast
  - Avoid unnecessary traffic
  - Avoid overloading marginal DNS servers
- CoDNS works as an insurance in which if the DNS is too slow the peers try to resolve the query before the local DNS
  - Minimal overhead if the system is function properly

Bootstrapping

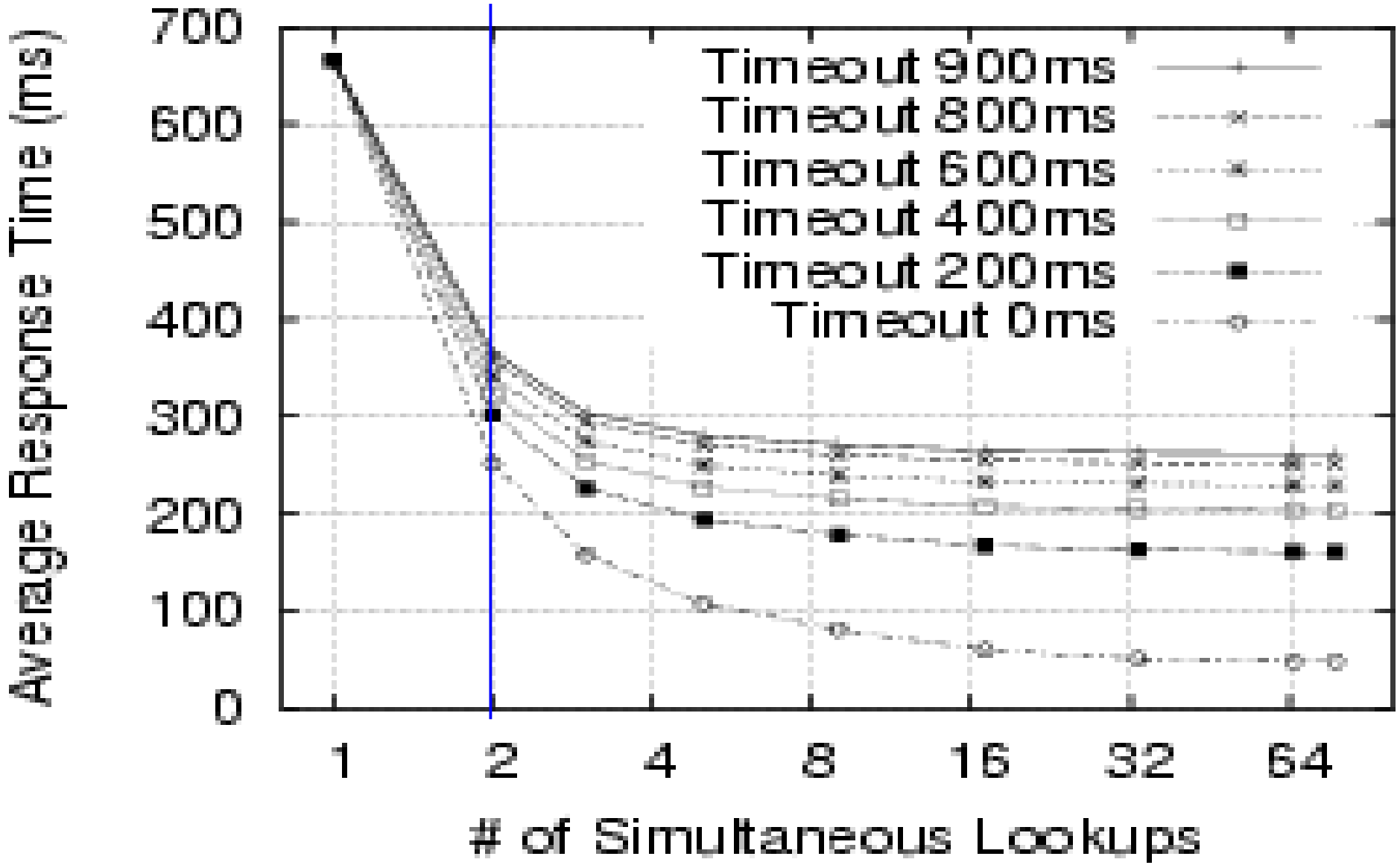
- CoDNS must resolve the names of the peers before it can start working.
- CoDNS resolves the name of the first peer by using local DNS and then it start using that peer to resolve other names

Results

- Improved Average Response Time

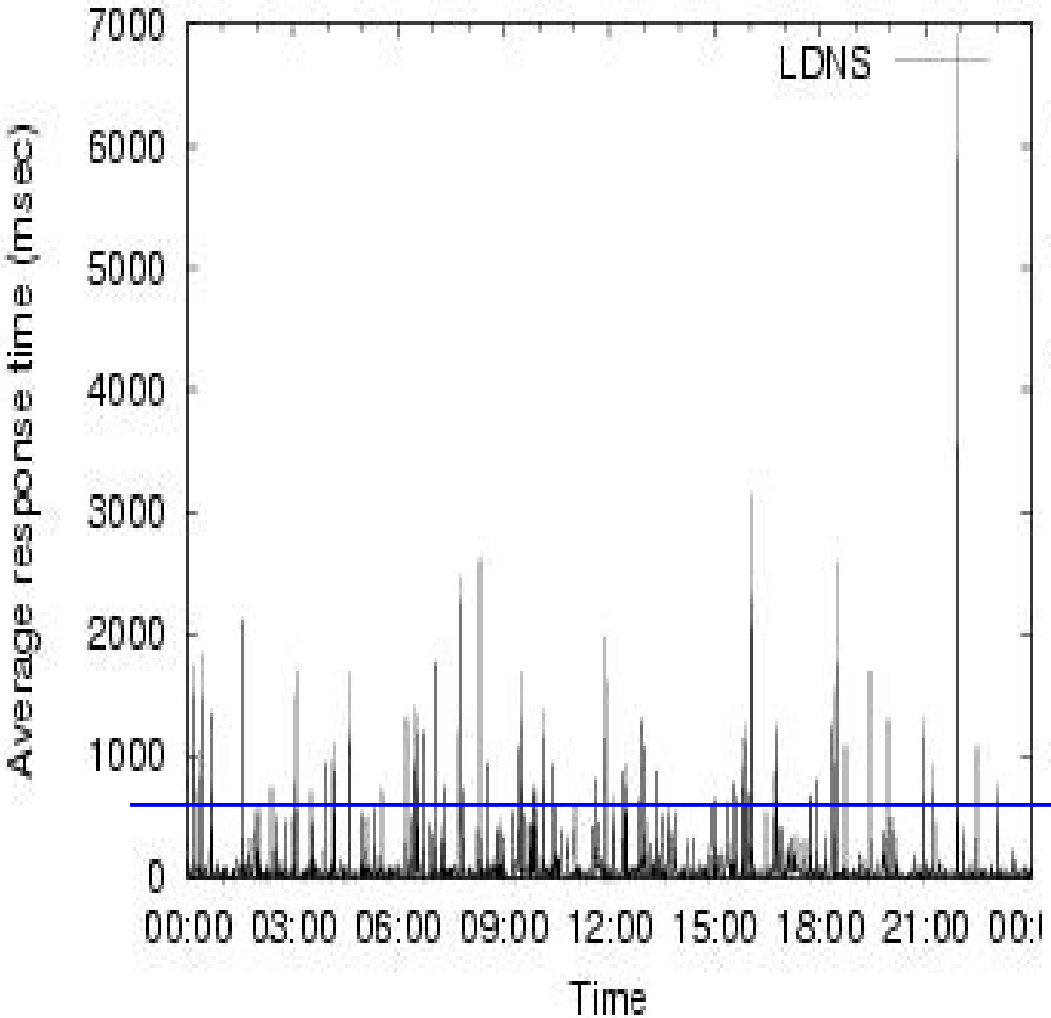
- Minimal Overhead
  - Even using only 1 extra lookup the average response time reduces by more than half
  - No overhead if the lookup is local (98% of the requests)

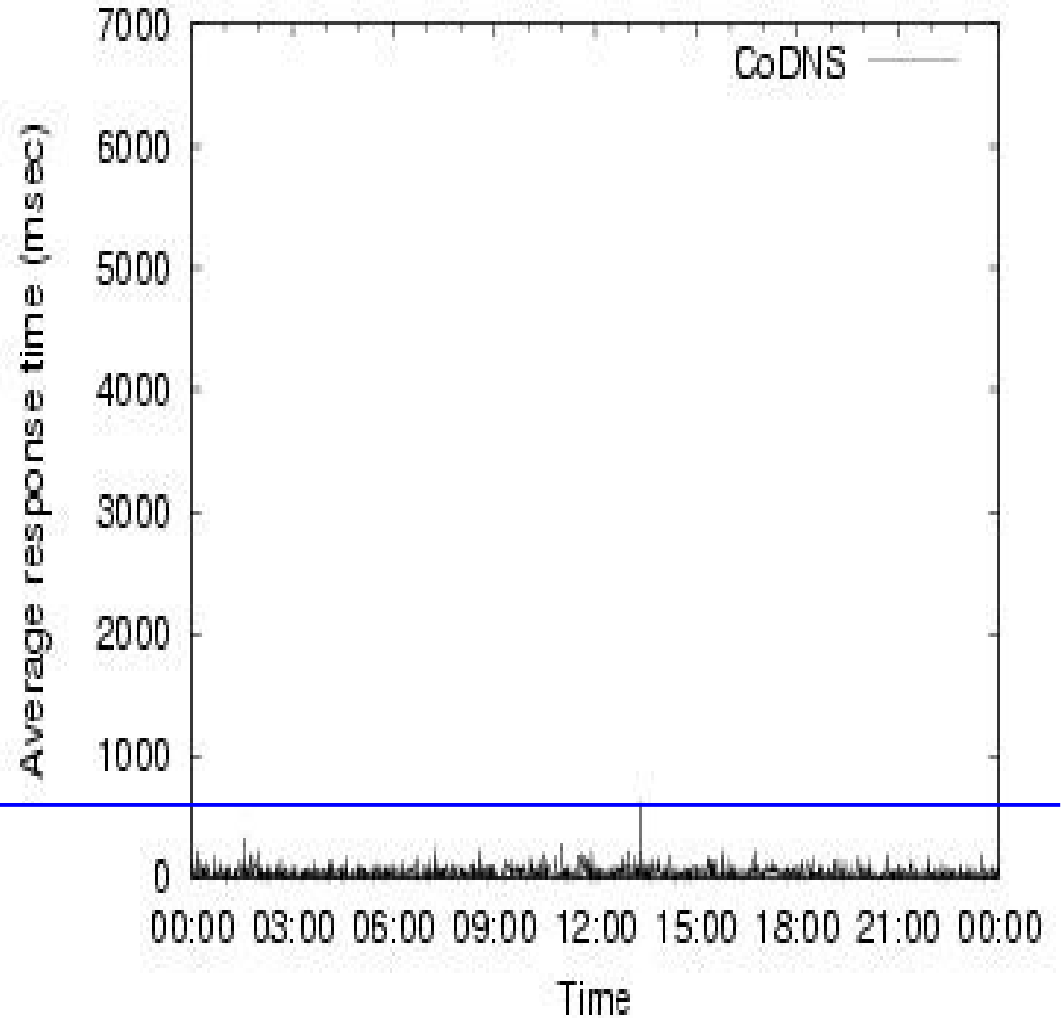
Results


Average Response Time (ms)
700
600
500
400
300
200
100
0
1
2
4
8
16
32
64
# of Simultaneous Lookups
Timeout 900ms
Timeout 800ms
Timeout 600ms
Timeout 400ms
Timeout 200ms
Timeout 0ms

Results

Minute granularity Average Response Time for one day on planetlab1.cs.cornell.edu





Results

- Improved Availability
  - 99.9% availability on 70% of the node (as opposed to 99% availability on 60% using DNS)

- Minimal adverse effect on CDN protocols

Criticism

- If a node gets compromised it can respond with a different address
- CoDNS relies on DNS to bootstrap
  - SEDNS in Rearchitecting DNS [Pfeifer *et. al*]
- Requires the deployment of a P2P component
  - Some other approaches obtain similar or better results using redundant lookups
  - CoDoNS [Ramasubramanian *et. Al*], SEDNS

Discussion

- If CoDNS was widely deployed. Do you think that in practice the system is prone to attacks from malicious peers?
- Is it preferable to use redundancy strategies over a P2P component?
- Is it feasible to overcome the limitation that CoDNS requires DNS to bootstrap?
- Other studies propose to replace DNS with a P2P (maybe Pastry). Would this be a better approach?

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

Anthony Rowston, Peter Druschel

Presented by Raoul Rivas and Chihung Lu

Pastry Overview

NodeId 10233102

Leaf set	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

- Peer to peer routing substrate
  - Efficiency, Scalability, Fault Resiliency, Self Organization
- Pastry can route to the numerically closest node for a given ID in O(log n)
- Pastry keeps a routing table a leaf set and a neighborhood set

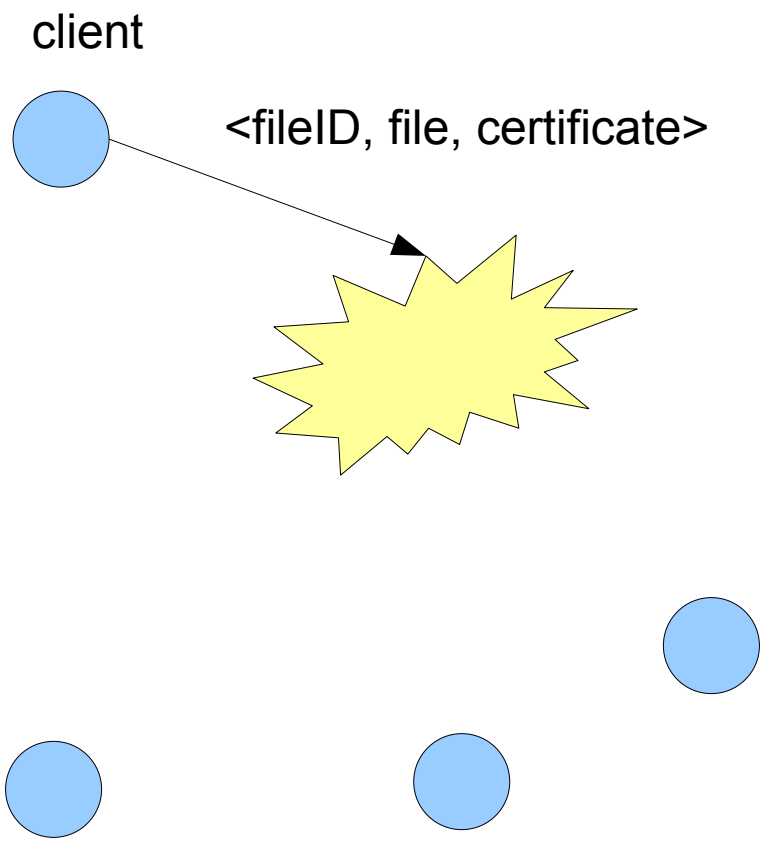
PAST Goals

- PAST is a P2P storage system aimed to achieve:
  - Global Storage
  - Strong Persistence
  - High Availability
  - Scalability
  - Security
- Note: PAST is deployed on top of Pastry but in theory it can be deployed on top of any other

PAST Operations

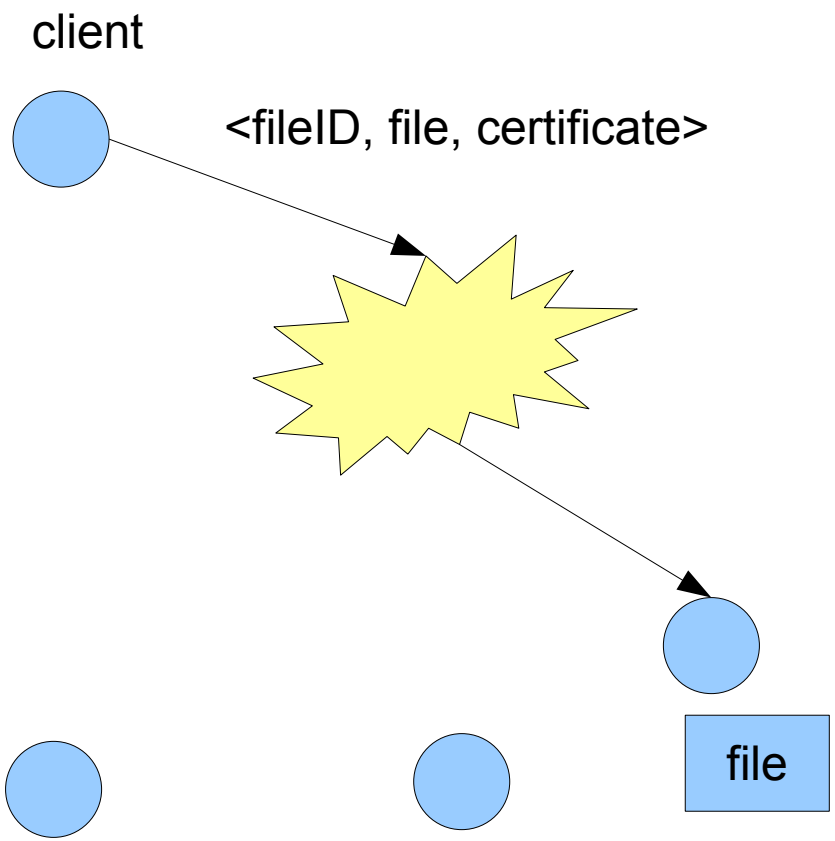
- fileID=Insert(name, credentials, k, file)
  - Stores a file in k different locations
  - Returns an unique identifier, making files immutable
- file=Lookup(fileID)
  - Retrieves a file from a close node
- Reclaim(fileID, credentials)
  - Delete but with weaker semantics
  - NO guarantee that it exists but it might be there
  - Avoid using agreement protocols

PAST Insert



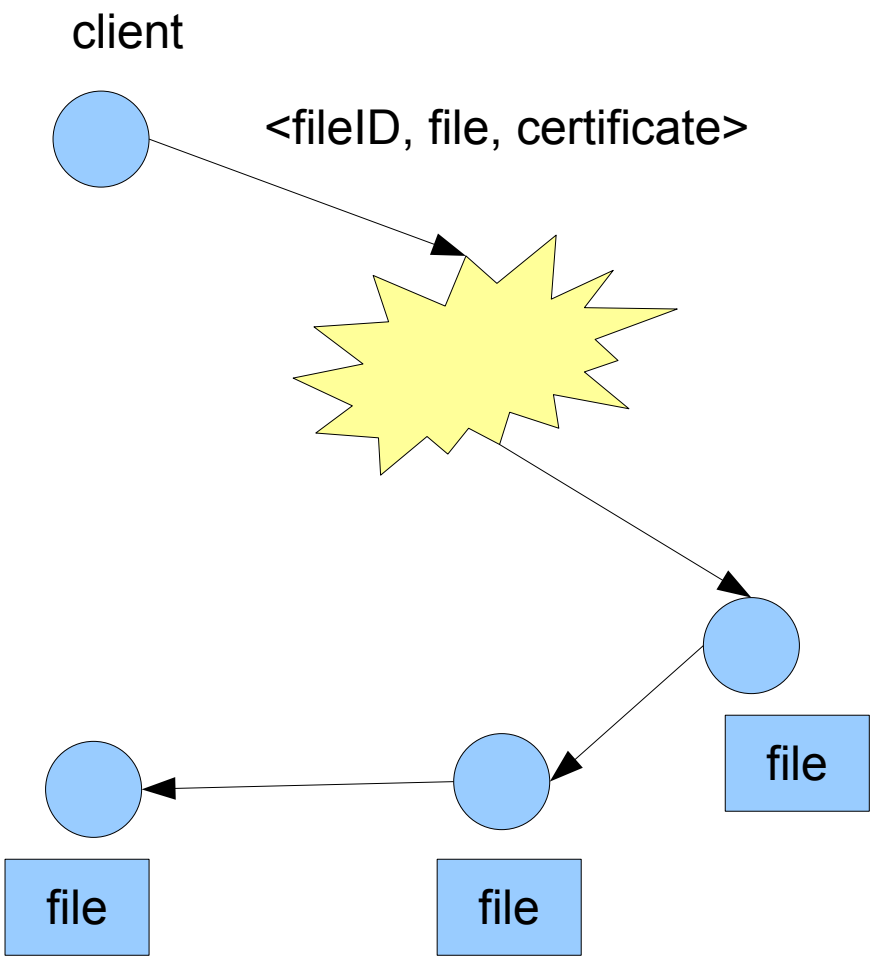
- Compute fileID using SHA
- Debit required storage from client quota
- Issue and sign a certificate with owners private key
- Route file via Pastry with destination fileID

PAST Insert



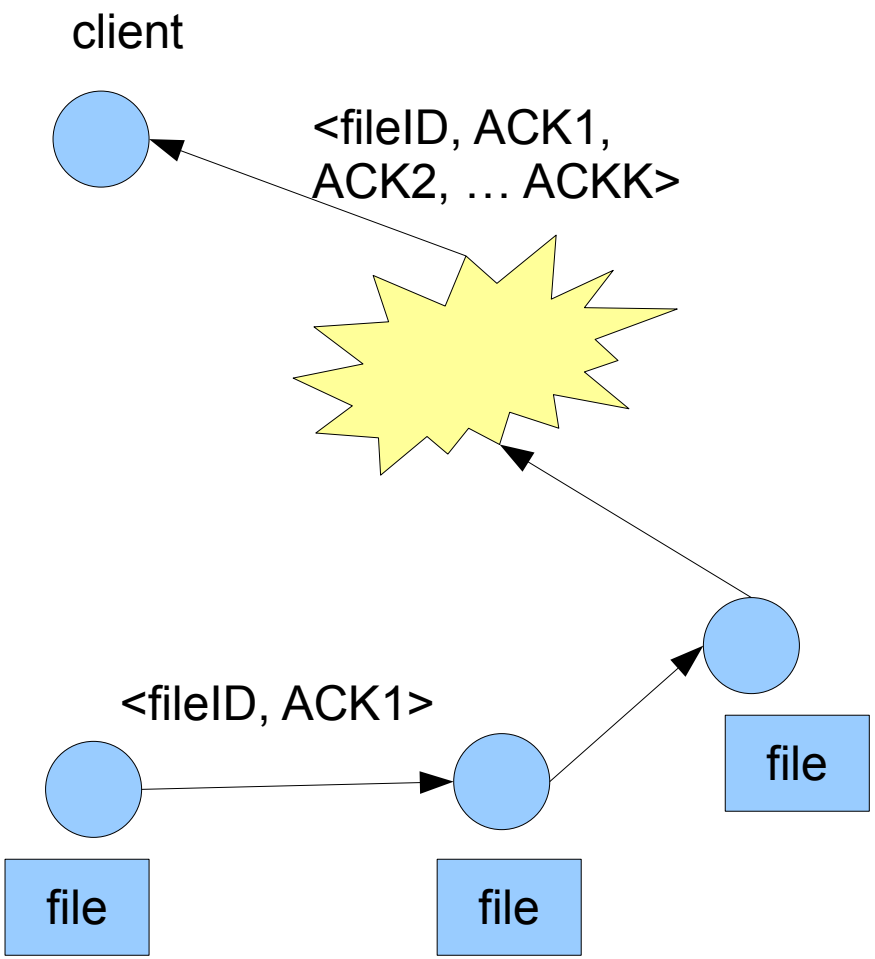
- When node reaches one of the k closests nodes:
- Verify certificate and checksum
- Insert the file and the certificate
- Forward it to the next k-1 nodes
- Passback Acknowledgement with store receipt to the client

PAST Insert



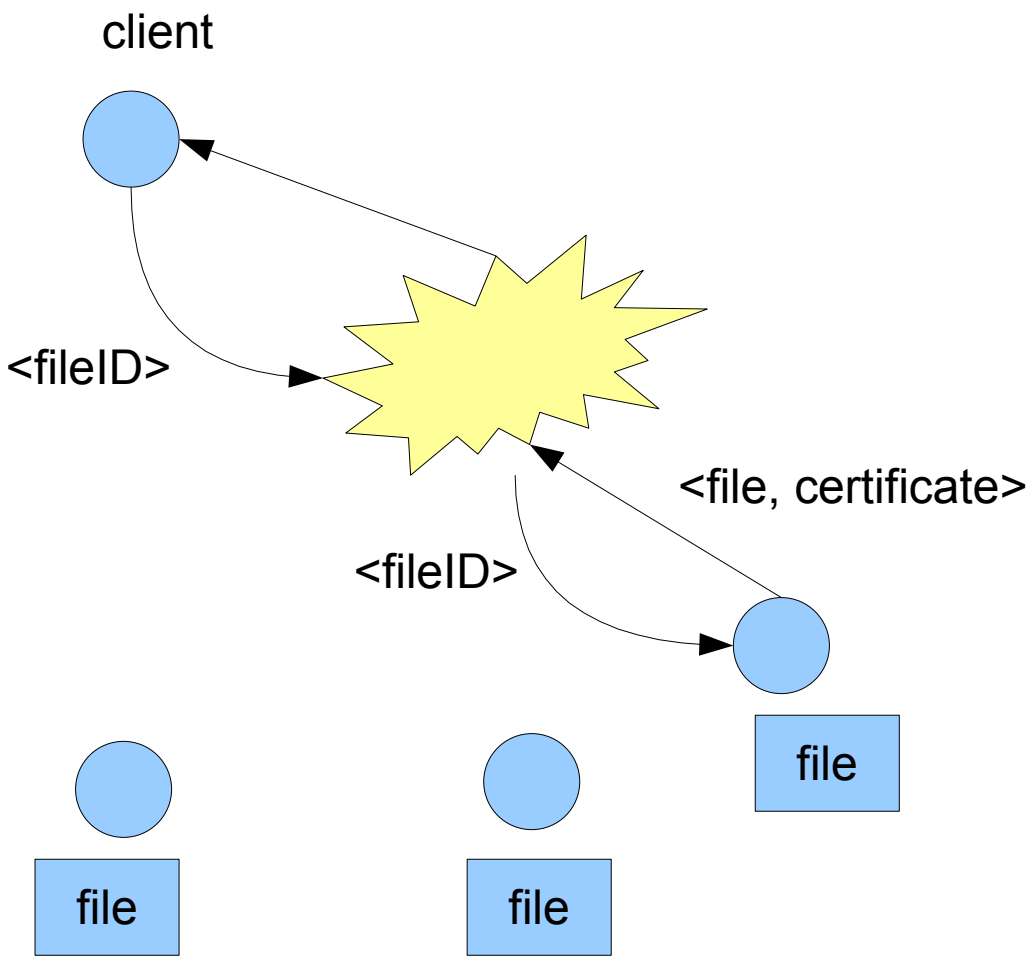
- When node reaches one of the k closests nodes:
- Verify certificate and checksum
- Insert the file and the certificate
- Forward it to the next k-1 nodes
- Passback Acknowledgement with store receipt to the client

PAST Insert



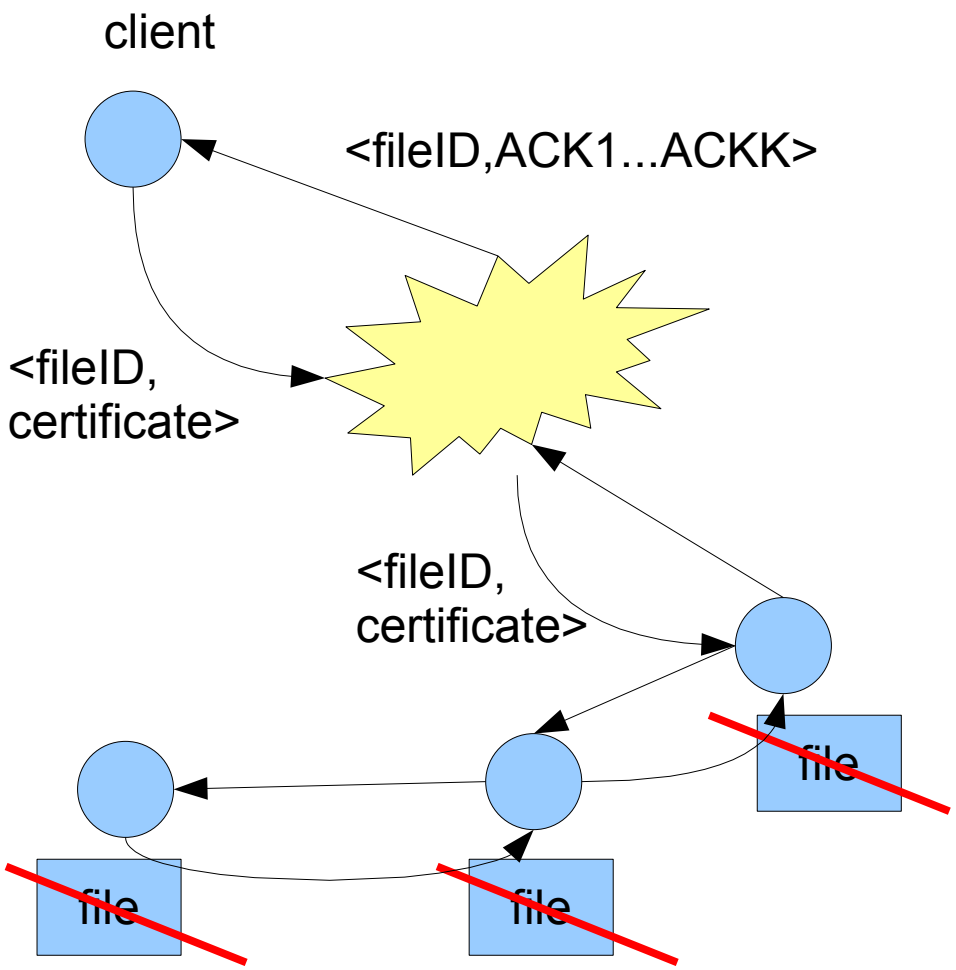
- When node reaches one of the k closests nodes:
- Verify certificate and checksum
- Insert the file and forward it to the next k-1 nodes
- Passback Acknowledgement with store receipt to the client

PAST Lookup



- Send request with fileID as destination
- When any node with the file receives the message reply with the certificate and the file
- DO NOT route the packet any further

PAST Reclaim

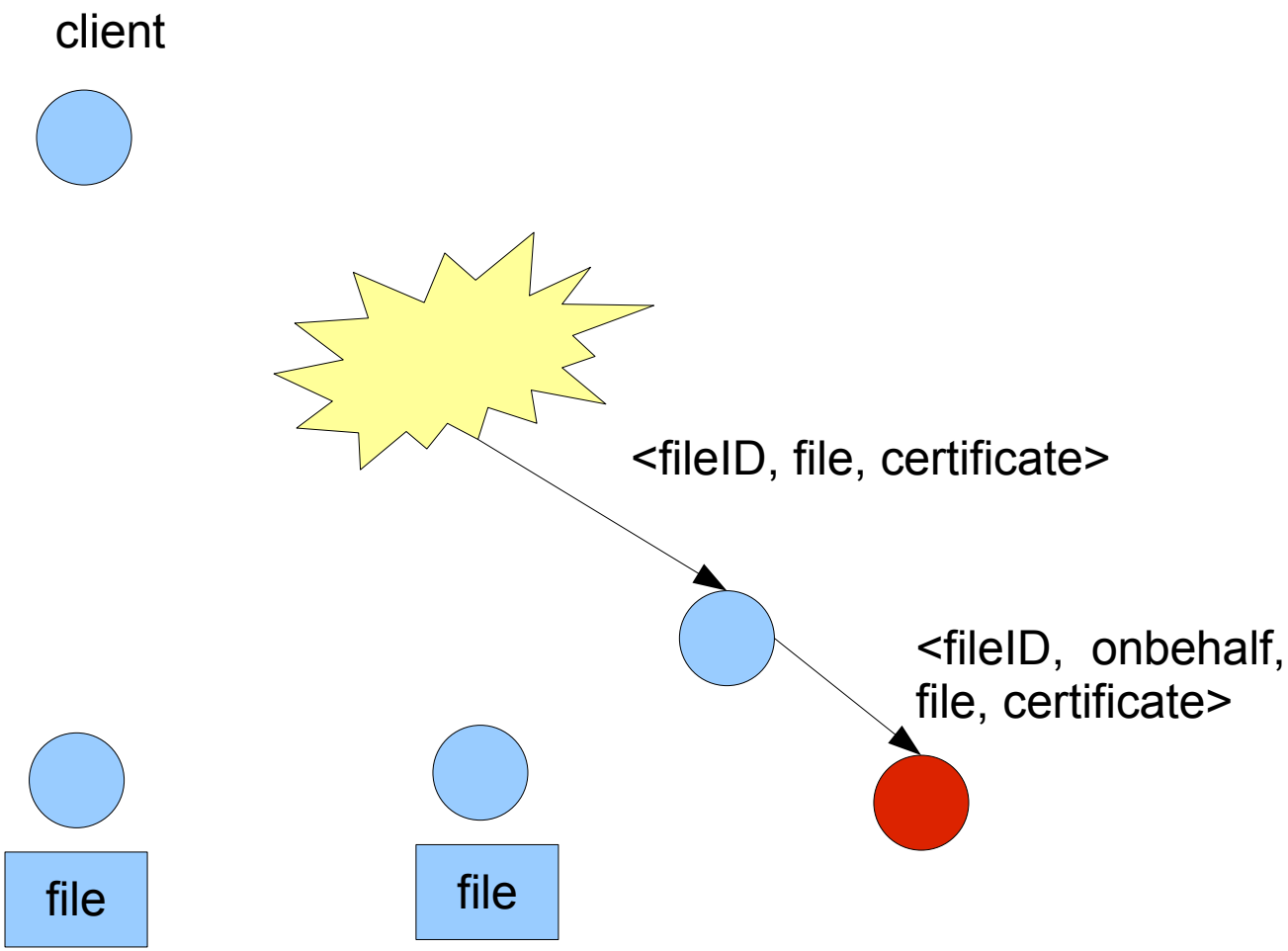


- Issue a reclaim certificate
- Send the fileID and the certificate to destination fileID
- Upon receive credit against client's quota
- Forward the request to the other k-1 nodes
- Passback a reclaim receipt to the client

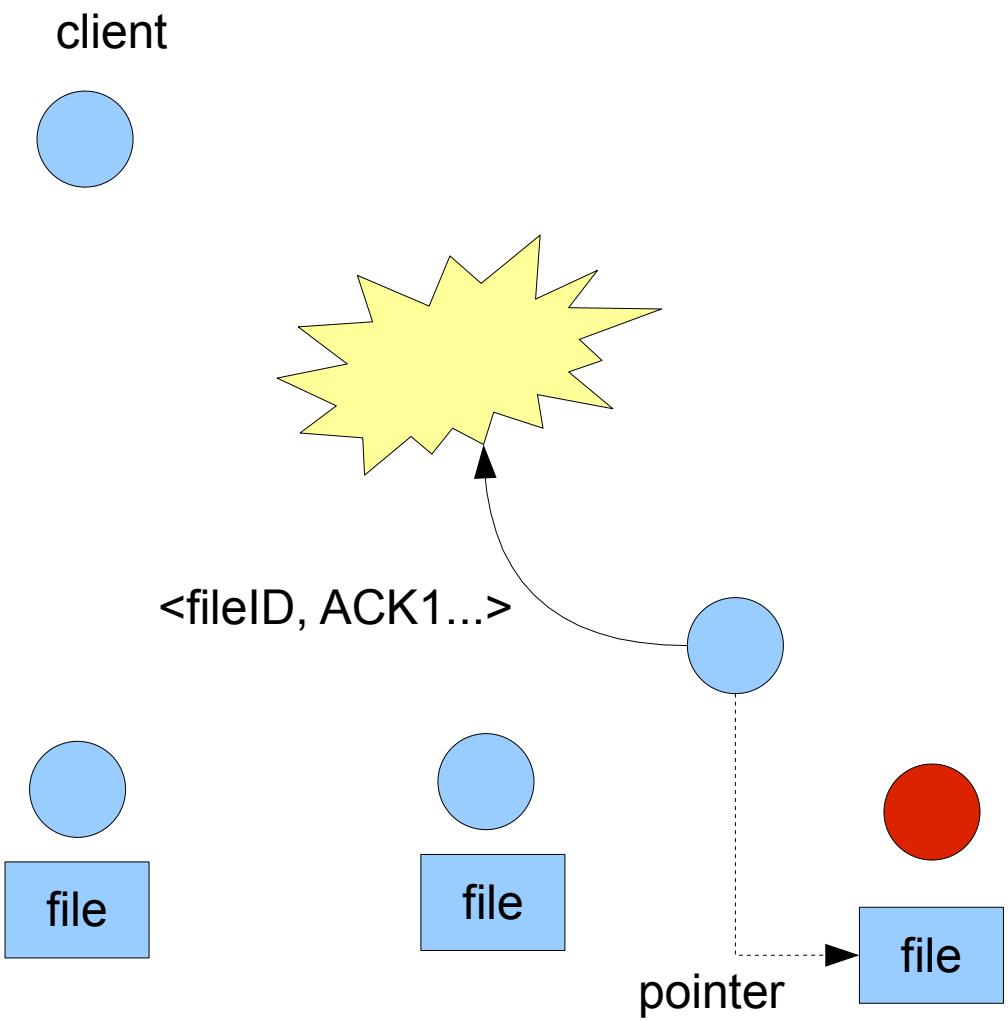
Replica Diversion

- It is a method used by PAST to maintain k copies of a file and to load balance the free space in the nodes of PAST under heavy utilization
- Procedure:
  - Try to insert in the corresponding node A, if not possible:
  - Choose a node B in the leaf set of A
  - Ask B to store the file in behalf of A
  - At node A, add a pointer to B
  - At node A issue a store receipt as usual

Replica Diversion



Replica Diversion



Insertion Policies

- When to accept a divert copy?
  - FileSize / FreeSpace_N > t_div
- When to accept a primary copy?
  - FileSize / FreeSpace_N > t_pri

Caching

- In PAST the free space is used to cache files
- Cached copies can e evicted and discarded at anytime
  - Improve performance
  - Cache performance degrades gracefully as the node utilization increases
- GreedyDual-Size replacement policy [Cao *et al*]
- Insert to the cache on routing

Results

- Replica diversion and file diversion are required to achive good utilization in PAST
  - With no replica and file diversion 51.1% of the insertions failed and the global utilization was 60.8%
- Leaf Set Size
  - When leaf set size is increased from 16 to 32 there is an increase in the performance of PAST.
    - Increased scope of the load balancing

Results

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 $t_{div} = 0.05$.

Results

- Increasing the value of t_pri (threshold of inserting primary copies) reduces the number of files that can be inserted in the nodes but increases the utilization
  - Many small files can be stored in the location of a large single file

t_{pri}	Succeed	Fail	File divers.	Replica divers.	Util.
0.5	88.02%	11.98%	4.43%	18.80%	99.7%
0.2	96.57%	3.43%	4.41%	18.13%	99.4%
0.1	99.34%	0.66%	3.47%	16.10%	98.2%
0.05	99.73%	0.27%	2.17%	12.86%	97.4%

Table 3: Insertion statistics and utilization of PAST as t_{pri} is varied and $t_{div} = 0.05$.

Results

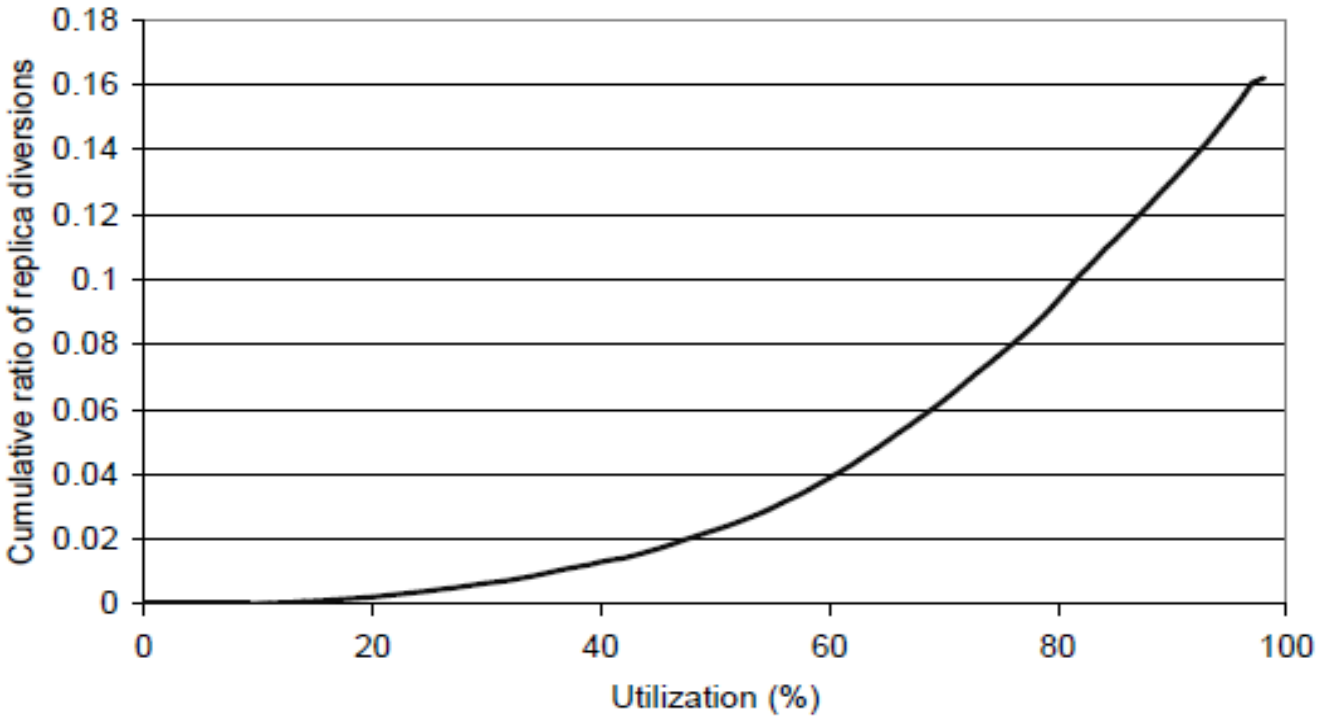


Figure 5: Cumulative ratio of replica diversions versus storage utilization, when $t_{pri} = 0.1$ and $t_{div} = 0.05$.

The number of diverted replicas remains small even at high utilization

Conclusions and Criticism

- PAST achieves global storage utilization of more than 98%.
- The percentage of failed insertions remains below 5% at even 95% storage utilization
- Failed insertions are biased toward large files
- PAST does not provide search capabilities

Discussion

- CFS is a block oriented storage system over Chord. Because CFS is block oriented it does not have the bias problem of failing to insert large files. However the overhead is higher. Is this acceptable? Which scheme is preferred and for which applications?
- Is it feasible to modify PAST to eliminate the problem of failing large insertions?