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

www.telmex.com.mx

www.telmex.com.mx

dns.uiuc.edu

NS a.ns.mx

a.root-servers.net

cairo.cs.uiuc.edu
DNS Overview

dns.uiuc.edu

www.telmex.com.mx

m.mx-ns.mx

www.telmex.com.mx

a.root-servers.net

a.ns.mx

cairo.cs.uiuc.edu
Motivation and Significance

- DNS failures can be divided into 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

![Diagram showing the process of lookups in CoDNS]

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

Diagram:
- Client
- App
- CoDNS Master process
  - List of Peers
  - CoDNS Slave process
  - CoDNS Slave process
- Query to DNS
- Query From other peers
- Query to other peers
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 starts 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
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

- 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 closest 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

client

=fileID, file, certificate=

file
PAST Insert

- When node reaches one of the k closest 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

Client

<fileID, onbehalf, file, certificate>

<fileID, file, certificate>

file

file

file
Replica Diversion

client
<fileID, ACK1...>

file
file
file

pointer

file
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 be 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 achieve 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

<table>
<thead>
<tr>
<th>Dist. Name</th>
<th>Succeed</th>
<th>Fail</th>
<th>File diversion</th>
<th>Replica diversion</th>
<th>Util.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>97.6%</td>
<td>2.4%</td>
<td>8.4%</td>
<td>14.8%</td>
<td>94.9%</td>
</tr>
<tr>
<td>$d_2$</td>
<td>97.8%</td>
<td>2.2%</td>
<td>8.0%</td>
<td>13.7%</td>
<td>94.8%</td>
</tr>
<tr>
<td>$d_3$</td>
<td>96.9%</td>
<td>3.1%</td>
<td>8.2%</td>
<td>17.7%</td>
<td>94.0%</td>
</tr>
<tr>
<td>$d_4$</td>
<td>94.5%</td>
<td>5.5%</td>
<td>10.2%</td>
<td>22.2%</td>
<td>94.1%</td>
</tr>
</tbody>
</table>

$l = 16$

<table>
<thead>
<tr>
<th>Dist. Name</th>
<th>Succeed</th>
<th>Fail</th>
<th>File diversion</th>
<th>Replica diversion</th>
<th>Util.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>99.3%</td>
<td>0.7%</td>
<td>3.5%</td>
<td>16.1%</td>
<td>98.2%</td>
</tr>
<tr>
<td>$d_2$</td>
<td>99.4%</td>
<td>0.6%</td>
<td>3.3%</td>
<td>15.0%</td>
<td>98.1%</td>
</tr>
<tr>
<td>$d_3$</td>
<td>99.4%</td>
<td>0.6%</td>
<td>3.1%</td>
<td>18.5%</td>
<td>98.1%</td>
</tr>
<tr>
<td>$d_4$</td>
<td>97.9%</td>
<td>2.1%</td>
<td>4.1%</td>
<td>23.3%</td>
<td>99.3%</td>
</tr>
</tbody>
</table>

$l = 32$

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

<table>
<thead>
<tr>
<th>$t_{pri}$</th>
<th>Succeed</th>
<th>Fail</th>
<th>File divers.</th>
<th>Replica divers.</th>
<th>Util.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>88.02%</td>
<td>11.98%</td>
<td>4.43%</td>
<td>18.80%</td>
<td>99.7%</td>
</tr>
<tr>
<td>0.2</td>
<td>96.57%</td>
<td>3.43%</td>
<td>4.41%</td>
<td>18.13%</td>
<td>99.4%</td>
</tr>
<tr>
<td>0.1</td>
<td>99.34%</td>
<td>0.66%</td>
<td>3.47%</td>
<td>16.10%</td>
<td>98.2%</td>
</tr>
<tr>
<td>0.05</td>
<td>99.73%</td>
<td>0.27%</td>
<td>2.17%</td>
<td>12.86%</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

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

Results

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?