Peer-to-Peer: Part II continued

Theophilus Benson
CS 0138 Spring ‘18

copyright@2018 Theophilus Benson, Thomas Doeppner and Rodrigo Foncesa
Today

Tapestry Continued: Adding nodes

OceanStore
Tapestry

- Assign each block a unique m-bit ID
  - crypto hash of its contents

- Assign each computer a unique m-bit ID

- Store multiple copies of blocks each at a number of computers
  - Use “salting”

- Store block addresses at computer that has closest ID: This NODE is ROOT
  - addresses are cached at other nodes

- Route requests for that block to the ROOT node
  - request is redirected to nearest computer that has copy of block
Publishing

Publishing == putting content into Tapestry

Content needs to be routed to the Root (node with closet ID)

Each object can have multiple root: use salting

- Step 1: pick random node
- Step 2: send publish request to random node
- Step 3: random node routes request to root (objectID)
- Step 4: root stores map of ObjectID to Client-IP
Tackling Root Failure

Each object can have multiple root: use salting
And if all roots fail?

Use soft state: state information times out

Must periodically refresh (for Tapestry, periodically republish)

**Client code:**

While (infinity)
  wait N seconds
  publish (objectID)

**Node code:**

publish event:
  object.lastPublish = now

While (infinity)
  if (object.lastPublish > now - threshold)
    delete Object
Tapestry Recap

• Each object has an ID
  • if you have an object —> periodically publish this object
    • if object not published, assumed lost
    • Publish == Route to Root
  • Multiple roots, one for each object

• IDs are stored at a Root node
  • Multiple root nodes exist for fault tolerance

• Routing: prefix based route
  • Each route gets you a digit closer to the destination
How do you add a Node?

- Steps for adding node n
  1) find existing node G (any random node)
  2) search for n’s hash starting at G (search for “root” for n)
  3) at each step i, fill in row i of n’s table with row i of table of node being visited
  4) stop when empty table entry is encountered
  5) fill in remainder of table with self entries
  6) notify other nodes to update their tables
### Route Table for Node: 3001

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXX</td>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td>3XXX</td>
<td>3001</td>
<td>3111</td>
<td>—</td>
<td>3312</td>
</tr>
<tr>
<td>30XX</td>
<td>3001</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>300X</td>
<td>—</td>
<td>3001</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### Route Table for Node: 3111

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXX</td>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3302</td>
</tr>
<tr>
<td>3XXX</td>
<td>—</td>
<td>3111</td>
<td>—</td>
<td>3312</td>
</tr>
<tr>
<td>31XX</td>
<td>—</td>
<td>3311</td>
<td>3120</td>
<td>—</td>
</tr>
<tr>
<td>311X</td>
<td>—</td>
<td>3311</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Step 4: Stop when empty table entry is encountered
Step 5: Fill in remainder of table with self entries
How do you add a Node?

Steps for adding node n
1) find existing node G (any random node)
2) search for n’s hash starting at G (search for “root” for n)
3) at each step i, fill in row i of n’s table with row i of table of node being visited
4) stop when empty table entry is encountered
5) fill in remainder of table with self entries
6) notify other nodes to update their tables
Notifying Others

• Need to insert n in all neighbor-table entries that are empty where n should go

• Before n was added, any search for n ended up at its root
Update Tables of other Nodes

- New node Broadcasts: send message to every one
- Old root sends to neighbors
  - Neighbors recursively send to other nights
- All nodes get updated: Too Expensive!!
Notifying Others

• Need to insert n in all neighbor-table entries that are empty where n should go

• Before n was added, any search for n ended up at its root

• Proceed backwards from root
  – each neighbor table includes back pointers to all nodes that route to it
  – flooding procedure:
    - on receipt of notification
      • if routing table contains hole where n goes
        – insert n
        – notify neighbors via back pointers
A Problem

- Node 3322 is added
- Object 3321’s surrogate was 3320
  - now it’s 3322
  - what about all the location info that was stored assuming 3320?
  - We need to “re-root” objects whose root is now 3322
### A Problem

Node 3322 is added

Object 3321’s surrogate was 3320
- now it’s 3322
- what about all the location info that was stored assuming 3320?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0121 128.148.158.13</td>
<td>1001 128.18.11.192</td>
<td>2130 128.113.225.127</td>
<td>3311 128.12.236.81</td>
</tr>
<tr>
<td>33xx</td>
<td>–</td>
<td>3311 128.12.236.81</td>
<td>3320 128.248.192.76</td>
<td>–</td>
</tr>
<tr>
<td>332x</td>
<td>3320 128.248.192.76</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Doing a Better Job …

• Find all nodes for which new node fills holes in neighbor tables
  – propagate new node on spanning tree of just the relevant nodes

• Handle “re-rooted” objects
  – efficiently …

• Build new neighbor tables
  – optimizing for closeness
Observation

- Let $\alpha$ be longest common prefix of new node and its root
- All nodes whose neighbor tables contain holes to be filled by new node have this prefix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0331 128.148.128.173</td>
<td>1332 128.138.117.92</td>
<td>2302 128.118.165.27</td>
<td>3312 128.213.97.6</td>
</tr>
<tr>
<td>3xx</td>
<td>3001 128.250.19.172</td>
<td>3111 128.172.53.237</td>
<td>–</td>
<td>3312 128.213.97.6</td>
</tr>
<tr>
<td>30xx</td>
<td>3001 128.250.19.172</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>300x</td>
<td>–</td>
<td>3001 128.250.19.172</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

n’s (3001’s) Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0331 128.148.128.173</td>
<td>1332 128.138.117.92</td>
<td>2302 128.118.165.27</td>
<td>3312 128.213.97.6</td>
</tr>
<tr>
<td>3xx</td>
<td>–</td>
<td>3111 128.172.53.237</td>
<td>–</td>
<td>3312 128.213.97.6</td>
</tr>
<tr>
<td>300x</td>
<td>–</td>
<td>3111 128.172.53.237</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

root’s (3111’s) Table
Application

• Send new node’s info to all nodes with prefix $\alpha$
  – hash and IP address

• How?
  – via spanning tree that reaches all such nodes
  – use “acknowledged multicast”
Acknowledged Multicast

\[
n.acknowledgedMulticast(\alpha, \text{function}) \{
  \text{if} \ (\text{notOnlyNodeWithPrefix}(\alpha))
  \text{for } i = 0 \text{ to } b-1
  \quad \text{neighbor} = \text{neighborWithPrefix}(\alpha \cdot i)
  \text{if} \ \text{neighbor exists}
  \quad S = \text{neighbor.acknowledgedMulticast}(\alpha \cdot i, \text{function})
  \text{else}
  \quad \text{apply function}
  \quad \text{wait } S
  \quad \text{SendAcknowledgement()}
\}
\]
Using Acknowledged Multicast

• To fill holes in neighbor tables with new node
  – supply function that does this

• To “re-root” object references (move them to the new node)
  – supply function that does this

• To get list of all nodes with given prefix
  – supply function that returns node IDs
Races

• Suppose a search for object x occurs while new node N is being added
  – N becomes the new root for x

• Potential problems:
  1) search arrives at N before object references are transferred to N
     1) request arrives at new root before object
  2) search arrives at old root after object references are transferred to N
     1) request arrives at old root after object moved
Problem 1 Solution

• Mark N as “being inserted”
  – on “object not found”
    - forward request to original root
Problem 2 Solution

• Include path with search request
  – on “object not found”
    - check neighbor table to see if path would still be taken
      • i.e., has hole subsequently been filled (by new root)?
      • if so, reroute to new root
Adding a Node: Chord V. Tapestry

• In both: you first ‘route’ to node
  • You re-root objects

• However
  • In Chord: you only update two nodes
  • In Tapestry: you update a lot of nodes
  • Differences in table sizes
    • Chord: update log(n)
    • fingers
    • Tapestry: update NxN matrix
Today

Tapestry Continued: Adding nodes

OceanStore
Peer to Peer III

Building a Global Storage Service
Today

• How to build a global data storage utility
  – Pooling resources from millions of devices that
    - Are not trusted
    - Come and go
    - Are independent (no centralized management)

• Target
  – $10^{10}$ users x $10^4$ files of $10^4$ bytes each = $10^{18}$ bytes
OceanStore
Goals

• Data lasts forever

• Data is tamper-proof and private
  – subject to user-specified authorization

• Access is generally quick
Issues

• What is the unit of replication
  – File or block?

• Finding a nearby copy

• Updating all copies
  – consistently

• Access control
• Integrity
• Fault tolerance
OceanStore

• Infrastructure
  – lots of administrative domains
  – servers trusted in aggregate, but not individually
    - arbitrary passive failures (crashes)
    - arbitrary active failures
      • really smart and malicious people and computers out there …
      • all communication is subject to eavesdropping and disruption
OceanStore

• Designed and developed at Berkeley, early 2000’s

• 5 Key Techniques
  – End-to-end encryption, *self-certifying* data
  – Tapestry self-organizing routing infrastructure
  – Erasure coding for durability (m-of-n)
  – Byzantine update commitment
  – Dynamic replica management

• Great use of a DOLR (Tapestry)
• *Teaser* for many other techniques we will see throughout the semester
Matching Issues to Techniques

Issues
- What is the unit of replication
- Finding a nearby copy
- Updating all copies
- Access control
- Integrity
- Fault tolerance

Techniques
- End-to-end encryption, self-certifying data
- Tapestry self-organizing routing infrastructure
- Erasure coding for durability (m-of-n)
- Byzantine update commitment
- Dynamic replica management
Caveats

• It’s far more complex than PuddleStore!

• This presentation is based on a number of OceanStore papers
  – not everything is totally clear
File Format

During update:
- Atomic change
- No transient state due to failure
Copy on Write (1)

(fileID)

Current Version

Indirect Block

Data Block 1

Data Block 2

Data Block 4

Indirect Block

Modified
Data Block 4

Modified
Data Block 3
Copy on Write (2)

- Change is atomic.
- Until version change.
  Reads go to hold data

```plaintext
fileID

Current Version

Indirect Block

Data Block 1

Data Block 2

Data Block 3

Data Block 4

Indirect Block

Modified Data Block 4

Modified Data Block 3
```
Block Management

Questions

• Where do the blocks go?

• How are they referenced?

• What if we can’t trust the computers that hold them?

• What if there are communication failures?

The Answers

• Tapestry
  – blocks are identified by secure hashes of their contents
  – contents may be encrypted if necessary
  – multiple copies may be published for redundancy

• Self-certifying blocks
  – What happens if the block changes?
Self Certifying Block

- BlockID = Hash (content)

- Ocean store
  - BlockID = Hash (encrypted(content))
  - Attacker can change the data
    - BlockID != Hash (attacker-data)
**OceanStore File**

- **BGUID**: block GUID
- **VGUID**: version GUID

**Definitions**

- **BGUID**: secure hash of a block of data
- **VGUID**: secure hash of the root block of a version

**Diagram**

- **VGUID**
- **Root Block**
- **Indirect Block**
- **Data Block**
- **Modified Data Block**
How do you find a file

- Need to go from file name to a record
- Actually just the pointer to the latest version
  - Only mutable record!
  - FileId is a hash of (file name, owner id)
Reading from Oceanstore

Get (BGUID) → Tapestry

VGUID

fileID

RootBlock

Indirect Block

Data Block 1

Modified Data Block 4

Modified Data Block 3

Modified Data Block 2

Modified Data Block 1
Reading from Oceanstore

Get (VGUID) → Tapestry

→ RootBlock

fileID

Current Version

fileID

Modified

RootBlock

Indirect Block

Data Block 1

Data Block 4

Data Block 3

Data Block 2