Implementation and Deployment of a Large-scale Network Infrastructure

Ben Y. Zhao
L. Huang, S. Rhea, J. Stribling, A. D. Joseph, J. D. Kubiatowicz
EECS, UC Berkeley
HP Labs, November 2002

Next-generation Network Applications

- Scalability in applications
  - Process/threads on single node server
  - Cluster (LAN): fast, reliable, unlimited comm.
  - Next step: scaling to the wide-area

- Complexities of global deployment
  - Network unreliability
    - BGP slow convergence, redundancy unexploited
  - Lack of administrative control over components
    - Constrains protocol deployment: multicast, congestion ctrl.
  - Management of large scale resources / components
    - Locate, utilize resources despite failures
Enabling Technology: DOLR  
(Decentralized Object Location and Routing)

A Solution

- Decentralized Object Location and Routing (DOLR)
  - wide-area overlay application infrastructure
    - Self-organizing, scalable
    - Fault-tolerant routing and object location
    - Efficient (b/w, latency) data delivery
  - Extensible, supports application-specific protocols

- Recent work
  - Tapestry, Chord, CAN, Pastry
  - Kademlia, Viceroy, …
What is Tapestry?

- **DOLR driving OceanStore global storage**
  (Zhao, Kubiatowicz, Joseph et al. 2000)

- **Network structure**
  - Nodes assigned bit sequence *nodeIds*
    namespace: $0-2^{160}$, based on some radix (e.g. 16)
  - *keys* from same namespace
    Keys dynamically map to 1 unique live node: *root*

- **Base API**
  - Publish / Unpublish (Object ID)
  - RouteToNode (NodeId)
  - RouteToObject (Object ID)

**Tapestry Mesh**
Incremental prefix-based routing

```
NodeID 0xEF97
      3
        
NodeID 0xEF32
      4
        
NodeID 0xEF37
      2
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF32
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF37
      4
        
NodeID 0xE932
      3
```

```
NodeID 0xEFBA
      3
        
NodeID 0xEF37
      2
        
NodeID 0x5032
      1
```

```
NodeID 0xE932
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF32
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF37
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF32
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF37
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF32
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF37
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF32
      4
        
NodeID 0xEF34
      1
```

```
NodeID 0xEF37
      4
        
NodeID 0xEF34
      1
```
Routing Mesh

- Routing via local routing tables
  - Based on incremental prefix matching
  - Example: 5324 routes to 0629 via
    5324 → 0231 → 0667 → 0625 → 0629
  - At $n^{th}$ hop, local node matches destination at least $n$ digits (if any such node exists)
  - $i^{th}$ entry in $n^{th}$ level routing table points to nearest node matching: $\text{prefix(local ID, n)+i}$

Properties
- At most $\log(N)$ # of overlay hops between nodes
- Routing table size: $b \times \log(N)$
- Actual entries have $c-1$ backups, total size: $c \times b \times \log(N)$

Object Location
Randomization and Locality
Object Location

- Distribute replicates of object references
  - Only references, not the data itself (level of indirection)
  - Place more of them closer to object itself
- Publication
  - Place object location pointers into network
  - Store hops between object and “root” node
- Location
  - Route message towards root from client
  - Redirect to object when location pointer found

Node Insertion

- Inserting new node $N$
  - Notify *need-to-know* nodes of $N$, $N$ fills null entries in their routing tables
  - Move locally rooted object references to $N$
  - Construct locally optimal routing table for $N$
  - Notify nearby nodes to $N$ for optimization
- Two phase node insertion
  - Acknowledged multicast
  - Nearest neighbor approximation
Acknowledged Multicast

- Reach need-to-know nodes of $N$ (e.g. 3111)
  - Add to routing table
  - Move root object references

Nearest Neighbor

- $N$ iterates: list = need-to-know nodes, $L = prefix (N, S)$
  - Measure distances of List, use to fill routing table, level $L$
  - Trim to $k$ closest nodes, list = backpointers from $k$ set, $L--$
  - Repeat until $L == 0$
Talk Outline

- Algorithms
- Architecture
  - Architectural components
  - Extensibility API
- Evaluation
- Ongoing Projects
- Conclusion

Single Tapestry Node

<table>
<thead>
<tr>
<th>Decentralized File Systems</th>
<th>Application-Level Multicast</th>
<th>Approximate Text Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Interface / Upcall API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Node Management</td>
<td>Routing Table &amp; Object Pointer DB</td>
<td>Router</td>
</tr>
<tr>
<td>Network Link Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transport Protocols</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Single Node Implementation

Applications

Enter/leave Tapestry

Dynamic Tapestry

State Maint.

Node Ins/del

Core Router

Routing Link Maintenance

Patchwork

Distance Map

UDP Pings

Network Stage

API calls

Upcalls

SED A Event-driven Framework

Java Virtual Machine

Message Routing

**Router: fast routing to nodes / objects**

Receive new location msg

Upcall?

no

Have Obj Pts

no

Forward to nextHop(h+1,G)

yes

Signal App

Upcall Handler

yes

Forward to nextHop(0,obj)

no

Receive new route msg

Upcall?

no

Forward to nextHop(h+1,G)

yes

Signal App

Upcall Handler
Extensibility API

- **deliver** \((G, A_{id}, \text{Msg})\)
  - Invoked at message destination
  - Asynchronous, returns immediately

- **forward** \((G, A_{id}, \text{Msg})\)
  - Invoked at intermediate hop in route
  - No action taken by default, application calls route()

- **route** \((G, A_{id}, \text{Msg}, \text{NextHopNodeSet})\)
  - Called by application to request message be routed to set of NextHopNodes

Local Operations

- Accessibility to Tapestry maintained state

- **nextHopSet = Llookup\((G, \text{Num})\)**
  - Accesses routing table
  - Returns up to num candidates for next hop towards \(G\)

- **objReferenceSet = Lsearch\((G, \text{num})\)**
  - Searches object references for \(G\)
  - Returns up to num references for object, sorted by increasing network distance
Deployment Status

- **C simulator**
  - Packet level simulation
  - Scales up to 10,000 nodes

- **Java implementation**
  - 50000 semicolons of Java, 270 class files
  - Deployed on local area cluster (40 nodes)
  - Deployed on Planet Lab global network (~100 distributed nodes)

Talk Outline

- Algorithms
- Architecture
- Evaluation
  - Micro-benchmarks
  - Stable network performance
  - Single and parallel node insertion

- Ongoing Projects
- Conclusion
Micro-benchmark Methodology

- Experiment run in LAN, GBit Ethernet
- Sender sends 60001 messages at full speed
- Measure inter-arrival time for last 50000 msgs
  - 10000 msgs: remove cold-start effects
  - 50000 msgs: remove network jitter effects

Micro-benchmark Results

- Constant processing overhead ~ 50μs
- Latency dominated by byte copying
- For 5K messages, throughput = ~10,000 msgs/sec
Large Scale Methodology

- Planet Lab global network
  - 98 machines at 42 institutions, in North America, Europe, Australia (~ 60 machines utilized)
  - 1.26Ghz PIII (1GB RAM), 1.8Ghz PIV (2GB RAM)
  - North American machines (2/3) on Internet2

- Tapestry Java deployment
  - 6-7 nodes on each physical machine
  - IBM Java JDK 1.30
  - Node virtualization inside JVM and SEDA
  - Scheduling between virtual nodes increases latency

Node to Node Routing

- Ratio of end-to-end routing latency to shortest ping distance between nodes
- All node pairs measured, placed into buckets

Median=31.5, 90th percentile=135
**Object Location**

- Ratio of end-to-end latency for object location, to shortest ping distance between client and object location
- Each node publishes 10,000 objects, lookup on all objects

![Object Location Graph]

**Latency to Insert Node**

- Latency to dynamically insert a node into an existing Tapestry, as function of size of existing Tapestry
- Humps due to expected filling of each routing level

![Latency to Insert Node Graph]
Bandwidth to Insert Node

- Cost in bandwidth of dynamically inserting a node into the Tapestry, amortized for each node in network
- Per node bandwidth decreases with size of network

Parallel Insertion Latency

- Latency to dynamically insert nodes in unison into an existing Tapestry of 200
- Shown as function of insertion group size / network size

90th percentile = 55042
Results Summary

- Lessons Learned
  - Node virtualization: resource contention
  - Accurate network distances hard to measure

- Efficiency verified
  - Msg processing = 50µs, Tput ~ 10,000msg/s
  - Route to node/object small factor over optimal

- Algorithmic scalability
  - Single node latency/bw scale sublinear to network size
  - Parallel insertion scales linearly with group size

Talk Outline

- Algorithms
- Architecture
- Evaluation
- Ongoing Projects
  - P2P landmark routing: Brocade
  - Applications: Shuttle, Interweave, ATA
- Conclusion
State of the Art Routing

- High dimensionality and coordinate-based P2P routing
  - Tapestry, Pastry, Chord, CAN, etc…
  - Sub-linear storage and # of overlay hops per route
  - Properties dependent on random name distribution
  - Optimized for uniform mesh style networks

Reality

- Transit-stub topology, disparate resources per node
- Result: Inefficient inter-domain routing (b/w, latency)
Landmark Routing on P2P

- **Brocade**
  - Exploit non-uniformity
  - Minimize wide-area routing hops / bandwidth

- **Secondary overlay on top of Tapestry**
  - Select super-nodes by admin. domain
    - Divide network into cover sets
  - Super-nodes form secondary Tapestry
    - Advertise cover set as local objects
  - brocade routes directly into destination’s local network, then resumes p2p routing
Applications under Development

- **OceanStore**: global resilient file store
- **Shuttle**
  - Decentralized P2P chat service
  - Leverages Tapestry for fault-tolerant routing
- **Interweave**
  - Keyword searchable file sharing utility
  - Fully decentralized, exploits network locality
- **Approximate Text Addressing**
  - Uses text fingerprinting to map similar documents to single IDs
  - Killer app: decentralized spam mail filter

For More Information

Tapestry and related projects (and these slides):
http://www.cs.berkeley.edu/~ravenben/tapestry

OceanStore:
http://oceanstore.cs.berkeley.edu

Related papers:
http://oceanstore.cs.berkeley.edu/publications
http://www.cs.berkeley.edu/~ravenben/publications

ravenben@eecs.berkeley.edu