Outline for Today’s Lecture

- What is a DHT? (review)
- DHT Geometry
- Three classes of DHT applications (with examples):
  - rendezvous
  - storage
  - routing
- Why DHTs?

A DHT in Operation: Peers

A DHT in Operation: Overlay

A DHT in Operation: put()
**A DHT in Operation: put()**

**A DHT in Operation: get()**

**Fundamental Requirement**

- All puts and gets for a particular key must end up at the same machine
  - Even in the presence of failures and new nodes (churn)
- This depends on the DHT routing algorithm (last time)
  - Must be robust and scalable

**Making Sense of the Mess**

- There are (too) many DHT routing algorithms
- They are very hard to compare
  - The basic idea drowning in idiosyncratic system details
    - one does caching, another doesn’t, etc.
  - Each is evaluated in terms of basic performance numbers
    - leads to “black-box” comparison of entire system, not of basic algorithm
- Goal:
  - Separate fundamental design choice from system details
  - Understand some basic differences between DHT routing schemes
Three Aspects of a DHT Design

- Geometry: the basic graph structure
  - Tree, hypercube, ring, butterfly De Bruijn

- Distance function: metric imposed on geometry
  - d(id1, id2) measures how far apart two identifiers are

- Algorithm: rules for selection
  - neighbors
  - routes

Two Fundamental Properties

- Flexibility of neighbor selection: (FNS)
  - number of possible neighbors in a geometry

- Flexibility of route selection: (FRS)
  - average number of next-hop choices for all destinations

- Fine print:
  - select neighbors so that paths are still $O(\log N)$
  - select routes so that each hop is closer (by metric) to destination

Basic Insight

- Geometry determines flexibility

- Flexibility determines performance
  - FRS and FNS lead to shorter paths
  - FRS leads to more reliable paths

Chord DHT has Ring Geometry

Chord Distance function captures Ring

- Nodes are points on a clock-wise Ring
- $d(id1, id2) = \text{length of clock-wise arc between ids}$
  - $=(id2 - id1) \mod N$

Neighbor selection flexibility allowed by Ring Geometry

- Chord algorithm picks $i$th neighbor at $2^i$ distance
- A different algorithm picks $i$th neighbor from $[2^i, 2^{i+1})$
Route selection flexibility allowed by Ring Geometry

- Chord algorithm picks neighbor closest to destination
- A different algorithm picks the best of alternate paths

<table>
<thead>
<tr>
<th>Flexibility</th>
<th>Ordering of Geometries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbors (FNS)</td>
<td>Hypercube &lt;&lt; Tree, XOR, Ring, Hybrid (logN) (2^i)</td>
</tr>
<tr>
<td>Routes (FRS)</td>
<td>Tree &lt;&lt; XOR, Hybrid &lt; Hypercube &lt; Ring (1) (logN/2) (logN/2) (logN)</td>
</tr>
</tbody>
</table>

How relevant is flexibility for DHT routing performance?

Analysis of Static Resilience

Two aspects of robust routing
- Dynamic Recovery: how quickly routing state is recovered after failures
- Static Resilience: how well the network routes before recovery finishes
  - captures how quickly recovery algorithms need to work
  - depends on FRS

Evaluation:
- Fail a fraction of nodes, without recovering any state
- Metric: % Paths Failed

Analysis of Overlay Path Latency

- Goal: Minimize end-to-end overlay path latency
  - not just the number of hops
- Both FNS and FRS can reduce latency
  - Tree has FNS, Hypercube has FRS, Ring & XOR have both

Evaluation:
- Using Internet latency distributions (see paper)
**Summary of Results**

- FRS matters for Static Resilience
  - Ring has the best resiliance
- Both FNS and FRS reduce Overlay Path Latency
- But, FNS is far more important than FRS
  - Ring, Hybrid, Tree and XOR have high FNS

**Conclusion**

- Routing Geometry is a fundamental design choice
  - Geometry determines flexibility
  - Flexibility improves resilience and proximity
- Ring has the greatest flexibility

**Two Important Distinctions**

- When talking about DHTs, must be clear whether you mean
  - Peers vs Infrastructure
  - Library vs Service
Peers or Infrastructure

- **Peer:**
  - Application users provide nodes for DHT
  - Example: music sharing, cooperative web cache
  - Easier to get, less well behaved

- **Infrastructure:**
  - Set of managed nodes provide DHT service
  - Perhaps serve many applications
  - Example: Planetlab
  - Harder to get, but more reliable

Library or Service

- **Library:** DHT code bundled into application
  - Runs on each node running application
  - Each application requires own routing infrastructure
  - Allows customization of interface
  - Very flexible, but much duplication

- **Service:** single DHT shared by applications
  - Requires common infrastructure
  - But eliminates duplicate routing systems
  - Harder to get, and much less flexible, but easier on each individual app

DHTs vs Unstructured P2P

- **DHTs good at:**
  - exact match for “rare” items

- **DHTs bad at:**
  - keyword search, etc. [can't construct DHT-based Google]
  - tolerating extreme churn

- **Gnutella etc. good at:**
  - general search
  - finding common objects
  - very dynamic environments

- **Gnutella etc. bad at:**
  - finding “rare” items

Three Classes of DHT Applications

- **Rendezvous, Storage, and Routing**

Rendezvous Applications

- Consider a pairwise application like telephony

- If A wants to call B (using the Internet), A can do the following:
  - A looks up B’s “phone number” (IP address of current machine)
  - A’s phone client contacts B’s phone client

- What is needed is a way to “look up” where to contact someone, based on a username or some other global identifier

Using DHT for Rendezvous

- Each person has a globally unique key (say 128 bits)
  - Can be hash of a unique name, or something else

- Each client (telephony, chat, etc.) periodically stores the IP address (and other metadata) describing where they can be contacted
  - This is stored using their unique key

- When A wants to “call” B, it first does a get on B’s key
Key Point

- The key (or identifier) is globally unique and static
- The DHT infrastructure is used to store the mapping between that static (persistent) identifier and the current location
  - DHT functions as a dynamic and flat DNS
- This can handle:
  - IP mobility
  - Chat
  - Internet telephony
  - DNS
  - The Web!

Using DHTs for the Web

Oversimplified:

- Name data with key
- Store IP address of file server(s) holding data
  - replication trivial!
- To get data, lookup key
  - If want CDN-like behavior, make sure IP address handed back is close to requester (several ways to do this)

Three Classes of DHT Applications

- Rendezvous
- Storage
- Routing

Storage Applications

- Rendezvous applications use the DHT only to store small pointers (IP addresses, etc.)
- What about using DHTs for more serious storage, such as file systems?
  - Storage, bandwidth and availability all scale with DHT

Example: CFS (DHash over Chord)

- Goal: serve a read-only file system
  - files can be modified by owner
- Publisher inserts file system into DHT
- CFS client looks like an NFS file system:
  - /cfs/7f23bd0092
- CFS client fetches data from the DHT

CFS Uses Tree of Blocks

A "pointer": Root contains DHT key of Directory

Directory block contains filename/blockID pairs

Root

Directory

File1

Dir2

File3
CFS Uses Self-authentication

Immutable block: (Content-Hash Block)
key = CryptographicHash(value)
encourages data sharing!

Mutable block: (Public-key Block)
key = Kpub
value = data + Sign[data],Kpriv

Most Blocks are Immutable

![Diagram of a tree structure showing root, mutable block, immutable blocks, and replicas at nodes.]

- This is a single-writer mutable data structure

Adding a File to a Directory

![Diagram of a file system structure showing root, mutable block, immutable blocks, and replicas at nodes.]

Data Availability via Replication

- DHash replicates each key/value pair at the nodes after it on the circle
- It's easy to find replicas
- Put(k,v) to all
- Get(k) from closest

First Live Successor Manages Replicas

![Diagram showing replica management and block localization.]

Example: OceanStore

- Project here at Berkeley: the uber file system
- Many facets to design, won’t cover them here
- Main contrast with CFS here:
  - CFS: stores files
  - OceanStore: stores pointers, deals with locality
Example: Usenet over a DHT

- Bulletin board (started in 1981)
  - Has grown exponentially in volume
  - 2004 volume is 1.4 Terabyte/day
- Hosting full Usenet has high costs
  - Large storage requirement
  - Bandwidth required: OC3+ (1 $30,000/month)
- Only 50 sites with full feed
- Goal: save Usenet news by reducing needed storage and bandwidth

Posting a Usenet Article

- User posts article to local server
- Server exchanges headers & article w. peers
- Headers allow sorting into newsgroups

UsenetDHT

- Store article in shared DHT
- Only “single” copy of Usenet needed
- Can scale DHT to handle increased volume
- Incentive for ISPs: cut external bandwidth by providing high-quality hosting for local DHT server

Usenet Architecture

- User posts article to local server
- Server writes article to DHT
- Server exchanges headers only
- All servers know about each article
UsenetDHT Tradeoff

- Distribute headers as before:
  - clients have local access to headers

- Bodies held in global DHT
  - only accessed when read
  - greater latency, lower overhead

UsenetDHT: potential savings

<table>
<thead>
<tr>
<th></th>
<th>Net bandwidth</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usenet</td>
<td>12 Megabyte/s</td>
<td>10 Terabyte/week</td>
</tr>
<tr>
<td>UsenetDHT</td>
<td>120 Kbyte/s</td>
<td>60 Gbyte/week</td>
</tr>
</tbody>
</table>

- Suppose 300 site network
- Each site reads 1% of all articles

Example: Email (ePost)

- Mission critical application
  - required extreme use of erasure codes (99.99% recovery from 15% of the data)

- Approach:
  - Mail split into components and stored independently (by content)
    - multiple copies stored only once
  - Folders represented by logs
    - each entry stored separately, in a chained data structure

- User associated with multicast group:
  - When user is online, announces location and data delivered

Three Classes of DHT Applications

Rendezvous, Storage, and Routing

“Routing” Applications

- Application-layer multicast
- Video streaming
- Event notification systems
- ...

DHT-Based Multicast

- Application-layer, not IP layer
- Single-source, not any-source multicast
- Easy to extend to anycast
Tree Formation

- Group is associated with key
- "root" of group is node that owns key
- Any node that wants to join sends message to root, leaving forwarding state along path
- Message stops when it hits existing state for group
- Data sent from root reaches all nodes
Challenges

- Repairing tree
- Balancing duties among peers
- Low-latency routing (proximity-based DHT routing)

Example: Internet-Scale Query Processing

- Superficial motivation:
  - Database joins implemented with hash tables so...
  - Distributed joins can be implemented with DHTs
  - Scaling: latency $O(\log n)$ while computation $O(n)$

PIER

- Range of operators
  - Joins, aggregation (routing!), recursive, continuous queries
- Intended targets:
  - Data "in the wild" (files sharing, net monitoring, etc.)
  - No need for ACID semantics, just best-effort
- Future: more sophisticated queries
  - Range searches, etc.
  - Prefix Hash Tree

What's the Fuss about DHTs?

Distributed Systems Pre-Internet

- Connected by LANs (low loss and delay)
- Small scale (10s, maybe 100s per server)
- PODC literature focused on algorithms to achieve strict semantics in the face of failures
  - Two-phase commits
  - Synchronization
  - Byzantine agreement
  - Etc.

Distributed Systems Post-Internet

- Very different context:
  - Huge scales (thousands if not millions)
  - Highly variable connectivity
  - Failures common
  - Organic growth
- Abandoned distributed strict semantics
  - Adaptive apps rather than "guaranteed" infrastructure
- Adopted pairwise client-server approach
  - Server is centralized (even if server farm)
  - Relatively primitive approach (no sophisticated dist. algms.)
  - Little support from infrastructure or middleware
Problems with Centralized Server Farms

- Weak availability:
  - Susceptible to point failures and DoS attacks

- Management overhead
  - Data often manually partitioned to obtain scale
  - Management and maintenance large fraction of cost

- Per-application design (e.g., GoogleOS)
  - High hurdle for new applications

- Don’t leverage the advent of powerful clients
  - Limits scalability and availability

The DHT Community’s Goal

Produce a common infrastructure that will help solve these problems by being:

- Robust in the face of failures and attacks
  - Availability solved

- Self-configuring and self-managing
  - Management overhead reduced

- Usable for a wide variety of applications
  - No per-application design

- Able to support very large scales, with no assumptions about locality, etc.
  - No scaling limits, few restrictive assumptions

The Strategy

Define an interface for this infrastructure that is:

- Generally useful for a wide variety of applications
  - So many applications can leverage this work
  - Research: building on the interface

- Can be supported by a robust, self-configuring, widely-distributed infrastructure
  - Addressing the many problems raised before
  - Research: supporting the interface

Hourglass Analogy

Two Crucial Design Decisions

1. Technology for infrastructure: P2P
   - P2P=symmetric and self-organizing
   - Take advantage of powerful clients
   - Robust

2. Choice of interface: Lookup and Hash Table
   - Lookup(key) returns IP of host that “owns” key
   - Put() / Get(): standard HT interface
   - Proven to be useful programming interfaces
DHT Layering

**Distributed application**

```
<table>
<thead>
<tr>
<th>put(key, data)</th>
<th>get(key)</th>
<th>data</th>
</tr>
</thead>
</table>

**Distributed hash table**

```
| lookup(key) | node IP address |

**Lookup service**

```
node    node    node
```

- Application may be distributed over many nodes
- DHT distributes data storage over many nodes

---

Scenarios for DHT Usage

Places where DHTs might have a competitive advantage

---

Scenario #1: Public Infrastructure

- Consider CiteSeer or other nonprofit systems:
  - Service is very valuable to community
  - No source of revenue

- How can it expand?
  - Not enough support for expanding centralized facility
  - But many institutions would donate remote use of their local machines

- System problem:
  - Coordinating donated distributed infrastructure

---

The DHT Approach

- DHTs are well-suited to such settings
  - Inherently distributed with general interface
  - Naturally provides rendezvous and data sharing

- Developers can focus on how to layer app on top of DHT library
  - Resilience, scaling, all taken care of by DHT

- Typical assumption for important services:
  - Server-like nodes with good network access

---

Examples

- **CiteSeer**
  - Replicate current service (OverCite), but with 10x performance improvement
  - Use additional capacity to provide new features (e.g., SmartSeer’s alerts)

- **Cooperative CDNs**
  - Coral allows universities to collaboratively handle “slashdot” workloads
  - Operational today with many users

- **UsenetDHT**
  - Allows cooperative institutions to share bandwidth load
  - Operational system with small feed running

---

Scenario #2: Scaling Enterprise Apps

- Enterprises rely on several crucial services
  - Email, backup, file storage

- These services must be
  - Scalable
  - Robust
  - Easy to deploy
  - Easy to manage
  - Inexpensive
The DHT approach

- Build all services on DHT interface
- DHT infrastructure:
  - Scalable (just add nodes, need not be local)
  - Robust
  - Easy to deploy
  - Easy to manage
  - Exploits inexpensive commodity components

Examples

- Email
  - ePOST (Rice)
- Backup
  - MIT
- File storage
  - OceanStore

Scenario #3: Supporting Tiny Apps

- Many apps could use DHT interface, but are too small to deploy one themselves
  - Small: user population, importance, etc.
- Such an application could use a DHT service
- OpenDHT is a public DHT service
  - Lecture on this next week...

Scenario #4: Super-Resilence

- DHTs are a natural way to build super-resilient services
- DHTs would be a natural candidate for the next generation name service, or other such crucial pieces of the infrastructure

Not Just for Applications

- DHTs resolve flat names scalably
  - We haven’t been able to do this before
- How would we redesign the Internet, now that we can resolve flat names?