DISTRIBUTED HASH TABLES

Building large-scale, robust distributed applications

Frans Kaashoek
kaashoek@lcs.mit.edu

P2P: an exciting social development

- Internet users cooperating to share, for example, music files
  - Napster, Gnutella, Morpheus, KaZaA, etc.
- Lots of attention from the popular press
  "The ultimate form of democracy on the Internet"
  "The ultimate threat to copy-right protection on the Internet"
- Many vendors have launched P2P efforts
What is P2P?

- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
- Typically many nodes, but unreliable and heterogeneous
Traditional distributed computing: client/server

- Successful architecture, and will continue to be so
- Tremendous engineering necessary to make server farms scalable and robust
Application-level overlays

- One per application
- Nodes are decentralized
- NOC is centralized

*P2P systems are overlay networks without central control*
(Potential) P2P advantages

- Allows for scalable incremental growth
- Aggregate tremendous amount of computation and storage resources
- Tolerate faults or intentional attacks
Example P2P problem: lookup

- At the heart of all P2P systems
Centralized lookup (Napster)

SetLoc(“title”, N4)
Publisher@N4
Key=“title”
Value=file data...

N1 N2 N3

DB

N9 N6 N7 N8

Client

Lookup(“title”)

Simple, but O(N) state and a single point of failure
Flooded queries (Gnutella)

Robust, but worst case $O(N)$ messages per lookup
Another approach: distributed hash tables

- Nodes are the hash buckets
- Key identifies data uniquely
- DHT balances keys and data across nodes
- DHT replicates, caches, routes lookups, etc.
Why DHTs now?

- Demand pulls
  - Growing need for security and robustness
  - Large-scale distributed apps are difficult to build
  - Many applications use location-independent data

- Technology pushes
  - Bigger, faster, and better: every PC can be a server
  - Scalable lookup algorithms are available
  - Trustworthy systems from untrusted components
# DHT is a good interface

<table>
<thead>
<tr>
<th>DHT</th>
<th>UDP/IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup(key) → data</td>
<td>Send(IP address, data)</td>
</tr>
<tr>
<td>Insert(key, data)</td>
<td>Receive (IP address) → data</td>
</tr>
</tbody>
</table>

- Supports a wide range of applications, because few restrictions
  - Keys have no semantic meaning
  - Value is application dependent
- Minimal interface
DHT is a good shared infrastructure

- Applications inherit **some** security and robustness from DHT
  - DHT replicates data
  - Resistant to malicious participants
- Low-cost deployment
  - Self-organizing across administrative domains
  - Allows to be shared among applications
- Large scale supports Internet-scale workloads
DHTs support many applications

- File sharing [CFS, OceanStore, PAST, ...]
- Web cache [Squirrel, ..]
- Censor-resistant stores [Eternity, FreeNet, ..]
- Event notification [Scribe]
- Naming systems [ChordDNS, INS, ..]
- Query and indexing [Kademlia, ..]
- Communication primitives [I3, ..]
- Backup store [HiveNet]
- Web archive [Herodotus]

*data is location-independent*
Cooperative read-only file sharing

- DHT is a robust block store
- Client of DHT implements file system
File representation: self-authenticating data

- DHT key for block is SHA-1 (content block)
- File and file systems form Merkle hash trees
DHT distributes blocks by hashing IDs

- DHT replicates blocks for fault tolerance
- DHT caches popular blocks for load balance
Historical web archiver

- Goal: make and archive a daily check point of the Web
- Estimates:
  - Web is about 57 Tbyte, compressed HTML+img
  - New data per day: 580 Gbyte
    128 Tbyte per year with 5 replicas
- Design:
  - 12,810 nodes: 100 Gbyte disk each and 61 Kbit/s per node
Implementation using DHT

- **DHT usage:**
  - Crawler distributes crawling load by hash(URL)
  - Crawler inserts Web pages by hash(URL)
  - Client retrieve Web pages by hash(URL)

- **DHT replicates data for fault tolerance**
Backup store

- Goal: backup on other user’s machines
- Observations
  - Many user machines are not backed up
  - Backup requires significant manual effort
  - Many machines have lots of spare disk space
- Using DHT:
  - Merkle tree to validate integrity of data
  - Administrative and financial costs are less for all participants
  - Backups are robust (automatic off-site backups)
  - Blocks are stored once, if key = sha1(data)
Research challenges

1. Scalable lookup
2. Balance load (flash crowds)
3. Handling failures
4. Coping with systems in flux
5. Network-awareness for performance
6. Robustness with untrusted participants
7. Programming abstraction
8. Heterogeneity
9. Anonymity

*Goal: simple, provably-good algorithms*
1. Scalable lookup

- Map keys to nodes in a load-balanced way
  - Hash keys and nodes into a string of digit
  - Assign key to “closest” node
- Forward a lookup for a key to a closer node
- Insert: lookup + store
- Join: insert node in ring

Examples: CAN, Chord, Kademlia, Pastry, Tapestry, Viceroy, ....
Chord’s routing table: fingers
Lookups take $O(\log(N))$ hops

- Lookup: route to closest predecessor
CAN: exploit $d$ dimensions

- Each node is assigned a zone
- Nodes are identified by zone boundaries
- Join: chose random point, split its zone
Routing in 2-dimensions

- Routing is navigating a $d$-dimensional ID space
  - Route to closest neighbor in direction of destination
  - Routing table contains $O(d)$ neighbors
- Number of hops is $O(dN^{1/d})$
2. Balance load

- Hash function balances keys over nodes
- For popular keys, cache along the path
Why Caching Works Well

- Only $O(\log N)$ nodes have fingers pointing to N20
- This limits the single-block load on N20
3. Handling failures: redundancy

- Each node knows IP addresses of next $r$ nodes
- Each key is replicated at next $r$ nodes
Lookups find replicas

- Tradeoff between latency and bandwidth [Kademlia]
4. Systems in flux

- Lookup takes $\log(N)$ hops
  If system is stable
  But, system is never stable!
- What we desire are theorems of the type:
  1. In the almost-ideal state, ....$\log(N)$...
  2. System maintains almost-ideal state as nodes join and fail
Half-life [Liben-Nowell 2002]

- Doubling time: time for $N$ joins
- Halving time: time for $N/2$ old nodes to fail
- Half life: MIN(doubling-time, halving-time)
Applying half life

• For any node $u$ in any P2P networks:
  If $u$ wishes to stay connected with high probability,
  then, on average, $u$ must be notified about $\Omega(\log N)$ new nodes per half life

• And so on, ...
5. Optimize routing to reduce latency

- Nodes **close** on ring, but **far away** in Internet
- Goal: put nodes in routing table that result in few hops and low latency
“close” metric impacts choice of nearby nodes

- Chord’s numerical close and table restrict choice
- Prefix-based allows for choice
- Kademlia’s offers choice in nodes and places nodes in absolute order: close \((a,b) = \text{XOR}(a, b)\)
Neighbor set

- From $k$ nodes, insert nearest node with appropriate prefix in routing table
- Assumption: triangle inequality holds
Finding $k$ near neighbors

1. Ping random nodes
2. Swap neighbor sets with neighbors
   • Combine with random pings to explore
3. Provably-good algorithm to find nearby neighbors based on sampling [Karger and Ruhl 02]
Finding nearest neighbor
[Karger and Ruhl 02]

- **Maintain a neighbor table:**
  \[ \text{entry}_i = k \text{ nodes in distance } 2^i r \]

- **Find nearest node**
  1. Ask nodes in entry \( i \) for its nodes in entry \( i \)
  2. Insert nearest in entry \( i+1 \)

- **Claim:** *algorithm will find the most nearby nodes with high probability*
  Triangle inequality holds
  Doubling property holds

- **Chord maintains finger and neighbor table**
6. Malicious participants

- Attacker denies service
  - Flood DHT with data
- Attacker returns incorrect data [detectable]
  - Self-authenticating data
- Attacker denies data exists [liveness]
  - Bad node is responsible, but says no
  - Bad node supplies incorrect routing info
  - Bad nodes make a bad ring, and good node joins it

*Basic approach: use redundancy*
Sybil attack [Douceur 02]

- Attacker creates multiple identities
- Attacker controls enough nodes to foil the redundancy

*Need a way to control creation of node IDs*
One solution: secure node IDs

- Every node has a public key
- Certificate authority signs public key of good nodes
- Every node signs and verifies messages
- Quotas per publisher
Another solution: exploit practical byzantine protocols

- A core set of servers is pre-configured with keys and perform admission control
- The servers achieve consensus with a practical byzantine recovery protocol [Castro and Liskov ’99 and ’00]
- The servers serialize updates [OceanStore] or assign secure node IDs [Configuration service]
A more decentralized solution: weak secure node IDs

- ID = SHA-1 (IP-address node)
  - Assumption: attacker controls limited IP addresses
- Before using a node, challenge it to verify its ID
Using weak secure node IDS

- Detect malicious nodes
  - Define verifiable system properties
    - Each node has a successor
    - Data is stored at its successor
  - Allow querier to observe lookup progress
    - Each hop should bring the query closer
    - Cross check routing tables with random queries
- Recovery: assume limited number of bad nodes
- Quota per node ID
7. Programming abstraction

- Blocks versus files
- Database queries (join, etc.)
- Mutable data (writers)
- Atomicity of DHT operations
Philosophical questions

• How decentralized should systems be?
  • Gnutella versus content distribution network
  • Have a bit of both? (e.g., OceanStore)

• Why does the distributed systems community have more problems with decentralized systems than the networking community?
  • “A distributed system is a system in which a computer you don’t know about renders your own computer unusable”
  • Internet (BGP, NetNews)
What are we doing at MIT?

- Building a system based on Chord
  - Applications: CFS, Herodotus, Melody, Backup store, R/W file system, ...
- Collaborate with other institutions
  - P2P workshop
  - Big ITR
- Building a large-scale testbed
  - RON, PlanetLab
Summary

- Once we have DHTs, building large-scale distributed applications is easy
  - Single, shared infrastructure for many applications
  - Robust in the face of failures and attacks
  - Scalable to large number of servers
  - Self configuring across administrative domains
  - Easy to program

- Let’s build DHTs .... stay tuned ....