Distributed Hash Tables

Hari Balakrishnan
6.829 Fall 2016
Goal: Building robust distributed systems

Developing software infrastructure to build large-scale, robust distributed systems
Traditional distributed computing: client/server

- Successful architecture (and will continue to be useful)
- But it’s expensive to make server farms scalable and robust
Problems

• Availability
  • Susceptible to communication outages, malicious attacks
  • Inability to cope with sudden load (flash crowds, aka “slashdot effect”)

• Management overhead
  • Building scalable server farms is a hard problem
  • High fraction of cost is in management and maintenance

• Per-application design
  • TCP/IP socket interface is too “low level”
  • New distributed applications reinvent wheels
Goal

Produce system infrastructure that is:

1. Robust in the face of failures, attacks, load
   - High availability ("many nines")
2. Self-assembling and self-managing
3. Usable for a wide variety of applications
4. Able to support very large scale
Strategy

- Inspired by “peer-to-peer” (P2P) computing
- Focus on massive replication: reliable system from unreliable components
- Key idea: A new general-purpose interface for applications called a distributed hash table (DHT)
- Benefit: Applications don’t have to solve each of these problems themselves
What is a P2P system?

- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
  - Clients and servers are usually indistinguishable
- Large number of (unreliable) nodes
What can P2P teach us about infrastructure design?

- Resistant to DoS and failures
  - Safety in numbers, no single point of failure
- Self-assembling
  - Nodes insert themselves into structure
  - No manual configuration or oversight
- Flexible: nodes can be
  - Widely distributed or colocated
  - Powerful hosts or low-end PCs
- Each peer brings a little bit to the dance
  - Aggregate is equivalent to a big distributed server farm behind a fat network pipe
But what general interface?

• Big challenge for P2P: finding content
  • Many machines, must find one that holds data
  • File sharing systems good at finding “hay”, but not “needles”

• Essential task: lookup(key)
  • Given key, find host that has data (“value”) corresponding to that key

• Higher-level interface: put(key,val)/get(key)
  • Easy to layer on top of lookup()
  • Allows application to ignore details of storage
  • Good for some apps, not for others
Data-centric network abstraction

- TCP provides a “conversation” abstraction
  socket = connect (IP address, port);
  send(data on socket); /* goes to IP addr / TCP port */

- A DHT provides a “data-centric” abstraction as an overlay over the Internet
  - A key is a semantic-free identifier for data
  - E.g., key = hash(filename)

```
pag(kev, value)
```

**Distributed application**

```
get (key)  ↓  value (data)
```

**DHT Infrastructure**
DHT layering

- Application may be distributed over many nodes
- DHT distributes the key-value data store over many nodes
- Many applications can use the same DHT infrastructure
Virtues of DHT Interface

• Simple and demonstrably useful
• put/get API supports wide range of apps
  • No structure/meaning imposed on keys
  • Scalable, flat name space
  • Location-independent names → easy to replicate and move keys (content)!
• Key/value pairs are persistent and global
  • Can store other keys (or other names or IP addresses) in DHT values
  • And thus build complex data structures
Some sample DHT applications

- Storage systems
  - Persistent backup store ("P2P backup")
  - Read/Write file systems
  - Cooperative source code repository (CVS)
- Content distribution
  - "Grassroots" Web replication & content distribution
  - Robust netnews (Usenet)
  - Resilient Web links, untangling the Web from DNS
  - Web archiver with timeline
- Communication
  - Handling mobility, multicast, indirection
  - Email spam control
  - Better firewalls and coping with NATs
  - Various naming systems
- Distributed database query processing; event notification
The DHT “hourglass”

- Many applications can share same DHT service
- Resource multiplexing
A DHT can solve several hard problems

1. Scalable lookup
2. Handling failures
3. Network-awareness for performance
4. Data integrity
5. Coping with arrivals/departures
6. Balance load (flash crowds)
7. Robustness with untrusted participants
8. Heterogeneity
9. Anonymity
10. Indexing

*Goal: simple, provably-good algorithms*
A DHT in Operation: Peers
A DHT in Operation: Overlay
A DHT in Operation: put()

put($K_1, V_1$)
A DHT in Operation: put()
A DHT in Operation: put()
A DHT in Operation: `get()`
A DHT in Operation: get()
Designing a good lookup algorithm

- Map every (conceivable) key identifier to some machine in the network
  - Store key-value on that machine
  - Update mapping/storage as items and machines come and go

- Note: User does not choose key location
  - Not really restrictive: key in DHT can be a pointer
Requirements

- Load balance
  - Want responsibility for keys spread “evenly” among nodes
- Low maintenance overhead
  - As nodes come and go
- Efficient lookup of key to machine
  - Fast response
  - Little computation/bandwidth (no flooding queries)
- Fault tolerance to sudden node failures
Consequences

• As nodes come and go, costs too much bandwidth to notify everyone immediately.
• So, nodes only aware of some subset of DHT: their neighbors.
• In particular, home node for key might not be a neighbor.
• So, must find right node through a sequence of routing hops, asking neighbors about their neighbors...
Maintenance

- As nodes come and go, maintain set of neighbors for each machine
  - Keep neighbor sets small for reduced overhead
  - Low degree
- Maintain routing tables to traverse neighbor graph
  - Keep number of hops small for fast resolution
  - Low diameter
Degree-Diameter Tradeoff

• Suppose machine degree $d$
  • Each neighbor knows $d$ nodes, giving $d^2$ at distance 2
  • Up to distance $h$, can reach $1 + d^2 + d^3 + \ldots + d^h \sim d^h$

• If $n$ nodes, need $d^h > n$ to reach all nodes
  • Therefore, $h > \log_d n$

• Consequences:
  • For $h = 2$ (two-hop lookup), need $d > \sqrt{n}$
  • With degree $d = 2$, get $h = \log_2 n$
Tradeoffs

• With larger degree, we can hope to achieve
  • Smaller diameter
  • Better fault tolerance

• But higher degree implies
  • More neighbor-table state per node
  • Higher maintenance overhead to keep neighbor tables up to date
Routing

- Low diameter is good, but not enough
- Item may be close: But how to find it?
- Need routing rules:
  - Way to assign each item to specific machine
  - Way to find that node by traversing (few) routing hops
Routing by Imaginary Namespace Geography

- Common principle in all DHT designs
- Map all (conceivable) keys into some abstract geographic space
- Place machines in same space
- **Assignment:** key goes to “closest” node
- **Routing:** guarantee that any node that is not the destination has some neighbor “closer” to the destination
  - Route by repeatedly getting closer to destination
The Chord algorithm

- Each node has 160-bit ID
- ID space is circular
- Data keys are also IDs
- A key is stored on the next higher node
- Good load balance
- Consistent hashing
- Easy to find keys slowly by following chain of successors

(N90 is responsible for keys K61 through K90)
Fast routing with a small routing table

- Each node’s **routing table** lists nodes:
  - ½ way around circle
  - ¼ way around circle
  - ...
  - next around circle
- The table is small:
  - At most $\log N$ entries
Chord lookups take $O(\log N)$ hops

• Every step reduces the remaining distance to the destination by at least a factor of 2

• Lookups are fast:
  • At most $O(\log N)$ steps
  • Can be made even faster in practice

Node N32 looks up key K19
Joining: linked list insert

1. Lookup(36)
2. N36 sets its own successor pointer

N36

K30
K38

N40

N25
Join (3)

3. Copy keys 26..36 from N40 to N36
Join (4)
[Done later, in stabilization]

4. Set N25’s successor pointer

Update other routing entries in the background
Correct successors produce correct lookups
Join and stabilization

// join a Chord ring containing node n'.
\texttt{n.join}(n')
\begin{itemize}
\item \texttt{predecessor} = \texttt{nil};
\item \texttt{successor} = \texttt{n'.find.successor}(n);
\end{itemize}

// called periodically. verifies n’s immediate
// successor, and tells the successor about n.
\texttt{n.stabilize()}
\begin{itemize}
\item \texttt{x} = \texttt{successor.predecessor};
\item \texttt{if} \ (x \in (n, \texttt{successor}))
\begin{itemize}
\item \texttt{successor} = x;
\item \texttt{successor.notify}(n);
\end{itemize}
\end{itemize}

// n' thinks it might be our predecessor.
\texttt{n.notify}(n')
\begin{itemize}
\item \texttt{if} \ (\texttt{predecessor} \texttt{is nil} \texttt{or} \ n' \in (\texttt{predecessor},n))
\begin{itemize}
\item \texttt{predecessor} = n';
\end{itemize}
\end{itemize}
Fault-tolerance with successor lists

- Each node knows about next \( r \) nodes on circle
- Each key is stored by the \( r \) nodes after “owner” on the circle
- If \( r = \Omega (\log N) \), lookups are fast even when \( P(\text{node failure}) = 0.5 \)
Redundancy Provides Failure Resilience

- 1000 DHT nodes
- Average of 5 runs
- 6 replicas for each key (less than log N)
- Kill fraction of nodes
- Then measure how many lookups fail
- All replicas must be killed for lookup to fail
- Lookups still return fast!

When 50% of nodes fail, only 0.12% of lookups fail!