LOCATING DATA ON THE NETWORK: CHORD

Mar 9, 2023
George Porter
ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Christo Wilson, NEU (used with permission)
  • Kyle Jamieson, Princeton
  • Tanenbaum and Van Steen, 3rd edition
ANNOUNCEMENTS

For Today: The Chord paper
For Tuesday: The DynamoDB paper
Two Examples of DHTS

- Chord
  - Fully decentralized
  - Over wide-area Internet
  - Designed for millions of end points

- DynamoDB
  - Managed within a single datacenter
  - Some centralization
  - 10s to 100s of end points
With consistent hashing, we have a way to distribute objects to a group of nodes

- With good load-balancing properties
- Such that when nodes come and go we don’t have to “re-hash” each object to a potentially new location
- But consistent hashing depends on knowing the set $S=\{S_0, S_1, \ldots, S_{n-1}\}$ of servers
- How do we do that?
IN A LOCAL NETWORK / DATACENTER

• We can just broadcast the set membership to each node
  • Local datacenter round-trip time is less than 1 millisecond
  • Number of nodes is likely in the 10s to 100s at most
• But what about bigger networks?
  • Distributed networks (e.g. geographically)
  • What about networks not managed by a single organization?
• Today we look at the Chord protocol
CHORD

- A distributed key-value store designed for Internet scale

- A truly distributed peer-to-peer protocol
  - Nodes join by simply knowing the IP address of any node in the network
  - No centralized organization or company manages it

- Aims for $O(1,000,000)$ nodes
  - Designed for sharing files, content, videos, music, etc.
CHORD LOOKUP ALGORITHM PROPERTIES

- **Interface:** lookup(key) → IP address

- **Efficient:** $O(\log N)$ messages per lookup
  - $N$ is the total number of servers
  - Note this isn’t a $O(1)$ DHT like SurfStore

- **Scalable:** $O(\log N)$ state per node

- **Robust:** survives massive failures
CHORD IDENTIFIERS

• **Key identifier** = SHA-1(key)

• **Node identifier** = SHA-1(IP address)

• SHA-1 distributes both uniformly

• *How does Chord partition data?*
  • *i.e.*, map key IDs to node IDs
- Assign $n$ tokens to random points on mod $2^k$ circle; hash key size = $k$
- Hash object to random circle position
- Put object in closest clockwise bucket
  - successor (key) → bucket

- Desired features –
  - Balance: No bucket has “too many” objects
  - Smoothness: Addition/removal of token minimizes object movements for other buckets
Key is stored at its successor: node with next-higher ID
CHORD: SUCCESSOR POINTERS
“Where is K80?”

“N90 has K80”

N90

N105

N120

N10

N32

N60

K80
**SIMPLE LOOKUP ALGORITHM**

```
Lookup(key-id)

succ ← my successor

if my-id < succ < key-id //next hop
    call Lookup(key-id) on succ

else //done
    return succ
```

- **Correctness** depends only on successors
IMPROVING PERFORMANCE

• **Problem:** Forwarding through successor is slow

• **Data structure is a linked list:** $O(n)$

• **Idea:** Can we make it more like a binary search?
  • Need to be able to halve distance at each step
• Skip Lists (Pugh, 1989)

• Consider a linked list:

• Lookup time: $O(n)$
• **Skip Lists (Pugh, 1989)**

• Consider a linked list:

  ![Diagram of a linked list with two rows of pointers]

  • Add 2\textsuperscript{nd} row of pointers spaced further apart
    • Still $O(n)$, but more efficient
    • Use 2\textsuperscript{nd} row to get as close as possible without going over
    • Then last row to get to the desired element
• Skip Lists (Pugh, 1989)
• Consider a linked list:

![](image)

• Add \(\log(N)\) rows
  • Get as close as possible on top row, then drop down a row, then drop down another row, until the bottom row
• \(O(\log N)\) lookup time
“FINGER TABLE” ALLOWS LOG N-TIME LOOKUPS

\[
\begin{align*}
1/4 & \quad 1/2 \\
1/8 & \\
1/16 & \quad 1/32 & \quad 1/64
\end{align*}
\]
FINGER $I$ POINTS TO SUCCESSOR OF $N+2^I$
IMPLICATION OF FINGER TABLES

- A binary lookup tree rooted at every node
  - Threaded through other nodes' finger tables

  - This is better than simply arranging the nodes in a single tree
    - Every node acts as a root
      - So there's no root hotspot
      - No single point of failure
      - But a lot more state in total
**Lookup with Finger Table**

\[ \text{Lookup}(\text{key-id}) \]

look in local finger table for highest \( n \): my-id < \( n < \) key-id

1. If \( n \) exists
   - call Lookup(key-id) on node \( n \) //next hop

2. Else
   - return my successor //done
THE CHORD RING \((2^5=32)\)
CHORD RING WITH SERVERS \{1,4,6,9,12,14,21,24,28\}
ADDING FINGER TABLES

4 + 2^0 = 4 + 1 = 5
4 + 2^1 = 4 + 2 = 6
4 + 2^2 = 4 + 4 = 8
4 + 2^3 = 4 + 8 = 12
4 + 2^4 = 4 + 16 = 20

12 + 2^0 = 12 + 1 = 13
12 + 2^1 = 12 + 2 = 14
12 + 2^2 = 12 + 4 = 16
12 + 2^3 = 12 + 8 = 20
12 + 2^4 = 12 + 16 = 28
Figure 5-4. Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.
AN ASIDE: IS LOG(N) FAST OR SLOW?

• For a million nodes, it’s 20 hops

• If each hop takes 50 milliseconds, lookups take a second

• If each hop has 10% chance of failure, it’s a couple of timeouts

• So in practice log(n) is better than O(n) but not great
JOINING: LINKED LIST INSERT

1. Lookup(36)
2. N36 sets its own successor pointer
JOIN (3)

3. Copy keys 26..36 from N40 to N36

N25

N40

N36

K30

K30

K38
NOTIFY MESSAGES MAINTAIN PREDECESSORS
“My predecessor is N36.”
JOINING: SUMMARY

- Predecessor pointer allows link to new node
- Update finger pointers in the background
- Correct successors produce correct lookups
WHAT CHORD GOT RIGHT

• **Consistent hashing**
  • Elegant way to divide a workload across machines
  • Very useful in clusters: actively used today in Amazon Dynamo and other systems

• **Replication** for high availability, efficient recovery after node failure

• **Incremental scalability**: “add nodes, capacity increases”

• **Self-management**: minimal configuration

• **Unique trait**: no single server to shut down/monitor