LOCATING DATA ON THE NETWORK: P2P NETWORKS, CHORD, AND DYNAMODB

Feb 17, 2022
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

Project 5 is out
TA video on project 5 coming soon

Today: Locating data on the network
Title: What we talk about when we talk about networking

Abstract: Networks, and the applications they support, sometimes treat each other as strangers. By shaking things up a bit—expressing networked systems as compositions of small, pure functions and making their dataflow a first-class consideration—we can often achieve friendlier couplings across the stack, to the benefit of performance, robustness, and understandability. This approach has proved helpful in several contexts: networking algorithms learned "in situ," feeding data from deployment back into training; real-time video conferencing, especially for musicians and actors during the pandemic; image compression in a distributed network filesystem; and a serverless computing framework that lets software burst briefly to 10,000 cores. In ongoing work, we're building a "functional" operating system that enforces a separation between IO (declared to the OS) and computation (reproducible by default). If this system can support a broad range of computational tasks with visibility into their dataflow, we envision a new service model for cloud computing: "computation as a service."

Bio: Keith Winstein is an assistant professor of computer science and, by courtesy, of electrical engineering at Stanford University. (https://cs.stanford.edu/~keithw)
LOCATING ITEMS (AT SCALE) IS A PRETTY HARD PROBLEM

• Consider our metadata store:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Version</th>
<th>hashlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>kitten.jpg</td>
<td>1</td>
<td>[h0, h1, h2, h3, h4]</td>
</tr>
<tr>
<td>puppy.mp4</td>
<td>1</td>
<td>[h5,h6,h7,h8,h9]</td>
</tr>
</tbody>
</table>

• Let’s figure out about how many files a single server metadata store can store...
LET’S CHOOSE AN AWS INSTANCE TYPE

- General Purpose
- Compute Optimized
- Memory Optimized
- Accelerated Computing
- Storage Optimized
- Instance Features
- Measuring Instance Performance
Amazon EC2 R6g instances are powered by Arm-based AWS Graviton2 processors. They deliver up to 40% better price performance over current generation R5 instances for memory-intensive applications.

Features:

- Custom built AWS Graviton2 Processor with 64-bit Arm Neoverse cores
- Support for Enhanced Networking with Up to 25 Gbps of Network bandwidth
- EBS-optimized by default
- Powered by the AWS Nitro System, a combination of dedicated hardware and lightweight hypervisor
- With R6gd instances, local NVMe-based SSDs are physically connected to the host server and provide block-level storage that is coupled to the lifetime of the instance
### MEMORY INSTANCE TYPES

<table>
<thead>
<tr>
<th>Instance Size</th>
<th>vCPU</th>
<th>Memory (GiB)</th>
<th>Instance Storage</th>
<th>Network Bandwidth (Gbps)**</th>
<th>EBS Bandwidth (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r6g.medium</td>
<td>1</td>
<td>8</td>
<td>EBS-Only</td>
<td>Up to 10</td>
<td>Up to 4,750</td>
</tr>
<tr>
<td>r6g.large</td>
<td>2</td>
<td>16</td>
<td>EBS-Only</td>
<td>Up to 10</td>
<td>Up to 4,750</td>
</tr>
<tr>
<td>r6g.xlarge</td>
<td>4</td>
<td>32</td>
<td>EBS-Only</td>
<td>Up to 10</td>
<td>Up to 4,750</td>
</tr>
<tr>
<td>r6g.2xlarge</td>
<td>8</td>
<td>64</td>
<td>EBS-Only</td>
<td>Up to 10</td>
<td>Up to 4,750</td>
</tr>
<tr>
<td>r6g.4xlarge</td>
<td>16</td>
<td>128</td>
<td>EBS-Only</td>
<td>Up to 10</td>
<td>4750</td>
</tr>
<tr>
<td>r6g.8xlarge</td>
<td>32</td>
<td>256</td>
<td>EBS-Only</td>
<td>12</td>
<td>9000</td>
</tr>
<tr>
<td>r6g.12xlarge</td>
<td>48</td>
<td>384</td>
<td>EBS-Only</td>
<td>20</td>
<td>13500</td>
</tr>
<tr>
<td>r6g.16xlarge</td>
<td>64</td>
<td>512</td>
<td>EBS-Only</td>
<td>25</td>
<td>19000</td>
</tr>
<tr>
<td>r6g.metal</td>
<td>64</td>
<td>512</td>
<td>EBS-Only</td>
<td>25</td>
<td>19000</td>
</tr>
</tbody>
</table>

Cost (per hour) of the r6g.16xlarge instance type: $3.2256
HOW MANY FILES CAN FIT INTO R6G.16XLARGE?

• 512GB of RAM

• Data requirements of each entry in the FileInfoMap?
  • Depends on size of the block...
  • Depends on distribution of file sizes...
    • Lots of small files? (e.g. C++, Java, Python, Go development)
    • Or big files? (audio or video files)

• Let’s see what the research literature says
File Size Distribution on UNIX Systems—Then and Now

Andrew S. Tanenbaum, Jorrit N. Herder*, Herbert Bos
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
{ast@cs.vu.nl, jnherder@cs.vu.nl, herbertb@cs.vu.nl}

Fig. 2. Data of Fig. 1 shown graphically.
Understanding Data Characteristics and Access Patterns in a Cloud Storage System

Figure 2. Bimodal file size distributions
A SIMPLE MODEL
BUT WHAT IF YOU NEED MORE SPACE?

• What if you have more than 20 million files??

• You need *scale*
SCALING

Vertical Scaling
(bigger machines)

Horizontal Scaling
(more machines)
VERTICAL SCALING

• Get a machine with more RAM, more storage, a faster CPU, more CPUs, ...

• Advantages:
  • Simple: Single machine abstraction
  • Simple: Only one IP address/hostname to consult

• Disadvantages:
  • Machines only get so big (have so much ram, etc)
  • What if the machine fails?
HORIZONTAL SCALING

• Form a *cluster* of 10, 100, 1000... servers that work together

• Advantages:
  • No one machine has to be very expensive/fancy
  • A failure of one machine doesn’t result in everything being lost

• Disadvantages:
  • How to find the data you’re looking for??
  • Performance is hard to reason about (subject of a future lecture, in fact)
HORIZONTAL SCALING ISSUES

- Probability of any failure in given period = $1-(1-p)^n$
  - $p =$ probability a machine fails in given period
  - $n =$ number of machines

For 50K machines, each with 99.99966% available
- 16% of the time, data center experiences failures

For 100K machines, failures 30% of the time!
THE LOCATION PROBLEM

• Given a cluster C of N servers, how do we locate the specific server Cᵢ responsible for a data item?

<table>
<thead>
<tr>
<th>Filename</th>
<th>Version</th>
<th>hashlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>kitten.jpg</td>
<td>1</td>
<td>[h₀, h₁, h₂, h₃, h₄]</td>
</tr>
<tr>
<td>puppy.mp4</td>
<td>1</td>
<td>[h₅,h₆,h₇,h₈,h₉]</td>
</tr>
</tbody>
</table>

• E.g. For a logical metadata storage service spread across N machines, which machine has the hash list for kitten.jpg? For puppy.mp4?
WHAT IS “FLAT” NAMING?

• The name doesn’t give you an indication of where the data is located

• Flat:
  • MAC address: 00:50:56:a3:0d:2a

• Vs hierarchical:
  • IP address: 206.109.2.12/24
  • DNS name: starbase.neosoft.com
FLAT NAME LOOKUP PROBLEM

Publisher (N4)

put("Shang-Chi.mov", [content])

N1

N2

N3

Internet

Client

get("Shang-Chi.mov")

N5

N6
CENTRALIZED LOOKUP (NAPSTER)

Publisher (N₄)

key="Shang-Chi.mov", value=[content]

SetLoc("Shang-Chi.mov", IP address of N₄)

N₁

N₂

N₃

DB

Lookup("Shang-Chi.mov")

Client

Simple, but O(N) state and a single point of failure
Outline

- Peer-to-peer networks
- Chord DHT
- DynamoDB DHT
PEER-TO-PEER (P2P) NETWORKS

- A distributed system architecture:
  - No centralized control
  - Nodes are roughly symmetric in function
  - Large number of unreliable nodes (could be reliable too)
FLOODED QUERIES (ORIGINAL GNUTELLA)

Robust, but $O(N = \text{number of peers})$ messages per lookup

key="Shang-Chi.mov", value=[content]
Can we make it robust, reasonable state, reasonable number of hops?
Outline

• Peer-to-peer networks
• Chord DHT
• DynamoDB DHT
SYSTEMATIC FLAT NAME LOOKUPS VIA DHTS

• Local hash table:
  
  \[
  \text{key} = \text{Hash(name)}
  \]
  
  \[
  \text{put(key, value)}
  \]
  
  \[
  \text{get(key)} \rightarrow \text{value}
  \]

• **Service:** Constant-time insertion and lookup

How can I do (roughly) this across millions of hosts on the Internet or within a giant datacenter application? Distributed Hash Table (DHT)
What is a DHT (and why)?

- Distributed Hash Table:
  
  \[
  \text{key} = \text{hash(data)}
  \]

  \[
  \text{lookup(key)} \rightarrow \text{IP addr}
  \]

  \[
  \text{send-RPC(IP address, put, key, data)}
  \]

  \[
  \text{send-RPC(IP address, get, key)} \rightarrow \text{data}
  \]

- Partitioning data in truly large-scale distributed systems
  - Tuples in a global database engine
  - Data blocks in SurfStore
  - Files in a P2P file-sharing system
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
• Consider problem of data partition:
  • Given object id \( X \), choose one of \( k \) servers to use

• Suppose instead we use modulo hashing:
  • Place \( X \) on server \( i = \text{hash}(X) \mod k \)

• What happens if a server fails or joins (\( k \leftarrow k\pm1 \))?  
  • or different clients have different estimate of \( k \)?
PROBLEMS WITH MODULO HASHING

Server

\[ h(x) = x + 1 \pmod{4} \]

Add one machine: \( h(x) = x + 1 \pmod{5} \)

All entries get **remapped** to new nodes!

\[ \rightarrow \text{Need to move objects over the network} \]
CHORD LOOKUP ALGORITHM PROPERTIES

- **Interface**: `lookup(key) → IP address`
- **Efficient**: $O(\log N)$ messages per lookup
  - $N$ is the total number of servers
- **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
CONSISTENT HASHING

- 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) $\rightarrow$ 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

Diagram showing successor pointers with nodes labeled N120, N105, N10, N32, N90, N60, K80.
“Where is K80?”

“N90 has K80”
**SIMPLE LOOKUP ALGORITHM**

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

\[
\begin{align*}
\text{succ} & \leftarrow \text{my successor} \\
\text{if } \text{my-id} & < \text{succ} < \text{key-id} \quad \text{// next hop} \\
& \quad \text{call Lookup(key-id) on succ} \\
\text{else} & \quad \text{// done} \\
\text{return} & \quad \text{succ}
\end{align*}
\]

- **Correctness** depends only on **successors**
CONSISTENT HASHING AND LOAD BALANCING

- Each node owns \(1/n^\text{th}\) of the ID space in expectation
  - Says nothing of request load per bucket

- If a node fails, its successor takes over bucket
  - **Smoothness goal ✔**: Only localized shift, not \(O(n)\)

- But now successor owns **two** buckets: \(2/n^\text{th}\) of key space
  - The failure has **upset the load balance**
VIRTUAL NODES

• **Idea:** Each **physical node** now maintains $v > 1$ tokens

  • Each token corresponds to a **virtual node**

• Each virtual node owns an expected $\frac{1}{(vn)^{th}}$ of ID space

• **Upon a physical node’s failure,** $v$ successors take over, each now stores $(v+1)/v \times 1/n^{th}$ of ID space

• **Result:** Better load balance with larger $v$
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:

![Linked List Diagram]

• Lookup time: $O(n)$
CHORD INTUITION

• Skip Lists (Pugh, 1989)
• Consider a linked list:

  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:

• 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
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**

**Lookup**(key-id)

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

if n exists

   call Lookup(key-id) on node n //next hop

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
NOTIFY MESSAGES MAINTAIN PREDECESSORS

- notify N25
- N25
- N40
- N36
- notify N36
“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 DHTS 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
UC San Diego