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

Feb 16, 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
Please read section 3 of the “Algorithmic nuggets in content delivery” paper from Akamai
• 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
Let’s pick the Arm-based Memory Instance

| R6g | R5  | R5a | R5b | R5n | R4  | X2gd | X1e | X1  | High Memory | z1d |

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

- File size distributions change over time and change depending on environment
- But to make it “easy”, let’s assume:
  - 90% of files are small (64KB) and 10% are big (256MB)
  - Let’s use a 64KB block size
    - 90% of files need 1 hash in the hash list
      - 64 bytes for hash + 64 for filename and version = 128 bytes
    - 10% need 4000 hashes in the hash list
      - 4000*64 for hash + 64 for filename and version = 256 KB
  - \[0.9 \times N \times 128 + 0.1 \times N \times 256 \text{ KB} = 512 \text{ GB}\]
  - \[N(0.9 \times 128 + 0.1 \times 256 \text{ KB}) = 512 \text{ GB}\]
  - \[N \text{ about } 1.997 \times 10^7 \text{ so just under 20 million}\]
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!
• Given a cluster \( C \) of \( N \) servers, how do we locate the specific server \( C_i \) responsible for a data item?

• E.g. For a logical metadata storage service spread across \( N \) machines, which machine has the hash list for kitten.jpg? For puppy.mp4?

<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>
PEER TO PEER NETWORKS

• A distributed set of nodes sharing content, often based on “flat” names, is called a peer to peer network.
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 ($N_4$)

$N_1$ $N_2$ $N_3$ $N_5$ $N_6$

Internet

get("LastOfUs.mov")

Client

put("LastOfUs.mov", [content])
CENTRALIZED LOOKUP (NAPSTER)

Publisher (N4) → DB
SetLoc("LastOfUs.mov", IP address of N4)

key="LastOfUs.mov", value=[content]

Client → DB
Lookup("LastOfUs.mov")

Simple, but O(N) state and a single point of failure
• 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="LastOfUs.mov", value=[content]
Can we make it robust, reasonable state, reasonable number of hops?
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:
  
  \[ \text{key} = \text{hash}(\text{data}) \]

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

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

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

- **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
SUMMARY OF IDEA

• We’re going to rely on hashing to map keys to servers
  • That way, to find a key (e.g. filename), just hash the name you’re looking for and consult just that server!
  • Cool... let’s see how that works in practice...
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!

→ Need to move objects over the network

We need a different hashing approach that doesn’t change everything when a server comes or goes…
CONSISTENT HASHING [KARGER ‘97]

- **Key identifier** $= \text{hash(key)}$

- **Node identifier** $= \text{hash(server’s IP address)}$ or $\text{hash(server’s hostname)}$ or $\text{hash(server’s identity)}$

- Same hash function maps two *different* types of data to the same ID space!
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
Figure 4: Consistent hashing first maps both objects and buckets (servers) to the unit circle. An object is then mapped to the next server that appears on the circle in clockwise order.
CONSISTENT HASHING [KARGER ‘97]

Key is stored at its successor: node with next-higher ID
CONSISTENT HASHING AND LOAD BALANCING

• Each node owns $\frac{1}{n}$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: $\frac{2}{n}$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 \( \frac{(v+1)}{v} \times \frac{1}{n^{th}} \) of ID space

• **Result**: Better load balance with larger \( v \)
Amazon’s DynamoDB data store is based on this concept of consistent hashing.