CS5412: DIVING IN: INSIDE THE DATA CENTER
We’ve seen one “cloud service”

- Inside a cloud, Dynamo is an example of a service used to make sure that cloud-hosted applications can scale at low cost.

- It offers a very scalable key-value way to store and fetch data, and this model is very important and popular, probably the single most important idea in all of cloud computing!

- But it isn’t the only example of a cloud service we should know about.
Once traffic reaches a data center it tunnels in

First passes through a filter that blocks attacks

Next, a router that directs it to a “tier one” computer (one of many) hosting the proper services

Each hosted company has its own “domain” but also runs inside a kind of virtual enterprise: looks like a private network, with its own IP addresses etc.

These can make use of shared services provided by the cloud provider, like file system services
Tier two and Inner Tiers

- If tier one faces the user and constructs responses, what lives in tier two?
  - Caching services are very common (many flavors)
  - Other kinds of rapidly responsive lightweight services that are massively scaled
  - Inner tier services might still have “online” roles, but tend to live on smaller numbers of nodes: maybe tens rather than hundreds or thousands
Tier two and Inner Tiers

- Tiers one and two soak up the load
  - This reduces load on the inner tiers, and in fact there may be further layers of caching to soak up load outside the data center (Akamai, Facebook edge, etc.)

- Many services use asynchronous streams of updates
  - Send updates “down” towards inner services but don’t wait and don’t hold locks!
  - Notifications percolate up, often showing up much later. Meanwhile, tier one runs with potentially out-of-data cached data.
Contrast with “Back office”

- A term often used for services and systems that don’t play online roles
  - In some sense the whole cloud has an outward facing side, handling users in real-time, and an inward side, doing “offline” tasks
  - Still can have immense numbers of nodes involved but the programming model has more of a batch feel to it

- For example, MapReduce (Hadoop)
Some interesting services we’ll consider

- Memcached: In-memory caching subsystem
- Dynamo: Amazon’s shopping cart
- BigTable: A “sparse table” for structured data
- GFS: Google File System
- Chubby: Google’s locking service
- Zookeeper: File system with locking, strong semantics
- TAO: Used by Facebook to track relationships such as “Friends” and “Likes”
- MapReduce: “Functional” computing for big datasets
Connection to DHT concept

- Last time we focused on a P2P style of DHT
- These services are mostly built as layers over a data center DHT deployment
  - Same idea and similar low-level functionality
  - But inside the data center we can avoid costly indirect routing. We’ll discuss that next time.
Memcached

Very simple concept:
- High performance distributed in-memory caching service that manages “objects”
- Key-value API has become an accepted standard
- Many implementations

Simplest versions: just a library that manages a list or a dictionary
Fanciest versions: distributed services implemented using a cluster of machines
Memcached defines a standard API

- Defines the calls the application can issue to the library or the server (either way, it looks like library)
- In theory, this means an application can be coded and tested using one version of memcached, then migrated to a different one

```python
function get_foo(foo_id)
    foo = memcached_get("foo:" . foo_id)
    if foo != null return foo
    foo = fetch_foo_from_database(foo_id)
    memcached_set("foo:" . foo_id, foo)
    return foo
end
```
A single memcached server is easy

- Today’s tools make it trivial to build a server
  - Build a program
  - Designate some of its methods as ones that expose service APIs
  - Tools will create stubs: library procedures that automate binding to the service
  - Now run your service at a suitable place and register it in the local registry

- Applications can do remote procedure calls, and these code paths are heavily optimized: quite fast
Can one use a cluster to host a scalable version of Memcached?

- This is what Amazon’s Dynamo service does!
- Built over a version of Chord DHT
  - Basic idea is to offer a key-value API, like memcached
  - But now we’ll have thousands of service instances
  - Used for shopping cart: a very high-load application
- Basic innovation?
  - To speed things up (think BASE), Dynamo sometimes puts data at the “wrong place”
  - Idea is that if the right nodes can’t be reached, put the data somewhere in the DHT, then allow repair mechanisms to migrate the information to the right place asynchronously
Dynamo in practice

- Suppose key should map to N56
- Dynamo replicates data on neighboring nodes (N1 here)
- Will also save key, value on subsequent nodes if targets don’t respond
- Data migrates to correct location eventually
Dynamo in practice

- When Amazon rolled Dynamo out, there was a huge need for scalable key-value storage, and Dynamo responded to this (e.g. shopping cart)

- But in fact it wasn’t popular with people more familiar with database APIs

- Eventually Amazon introduced Dynamo-DB which has a “NoSQL” API: SQL but with weak consistency. This has been far more successful for many uses.
Lessons learned?

- Notice that we started with Chord: a DHT for big P2P uses. But nobody has big P2P systems!

- Dynamo was initially just Chord adapted for use inside a data-center, as a service, adjusted to deal with the peculiar failure patterns seen at Amazon.

- But ultimately, with Dynamo-DB, Amazon was forced to change the “model” to bridge to developers.
BigTable

- Yet another key-value store!
- Built by Google over their GFS file system and Chubby lock service
- Idea is to create a flexible kind of table that can be expanded as needed dynamically
- Like Dynamo-DB, starts with a DHT idea but then grew to become a specialized solution with many features
Data model: a big map

- `<Row, Column, Timestamp>` triple for key Arbitrary “columns” on a row-by-row basis
  - Column family: qualifier. Family is heavyweight, qualifier lightweight
  - Column-oriented physical store- rows are sparse!
- Does not support a relational model
  - No table-wide integrity constraints
  - No multirow transactions
API

- **Metadata operations**
  - Create/delete tables, column families, change metadata

- ** Writes (atomic)**
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row

- **Reads**
  - Scanner: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns
Versions

- Data has associated version numbers
  - To perform a transaction, create a set of pages all using some new version number
  - Then can atomically install them

- For reads can let BigTable select the version or can tell it which one to access
How did they build it

- We’ll skim very lightly over the main data structures but you don’t need to learn the details.

- Idea is just to have a feeling for how it works.

- And how does it work? They map BigTable down to a kind of key-value system!
SSTable

- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is of block ranges, not values
Tablet

- Contains some range of rows of the table
- Built out of multiple SSTables

Tablet | Start: aardvark | End: apple
---|---|---
64K block | 64K block | 64K block
SSTable | Index

Index

64K block | 64K block | 64K block
SSTable | Index

Index
Table

- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap
Finding a tablet

- Stores: Key: table id + end row,  Data: location
- Cached at clients, which may detect data to be incorrect
  - in which case, lookup on hierarchy performed
- Also prefetched (for range queries)
Finding a tablet

- Stepping back, can you see how this is a bit like using a DHT in two layers?

- To look for something, you use the row and column information as a key, and that key lets you find the metadata on the tablet server, and then you can track down the actual data.

- So the DHT (key-value) concept can be “turned into” this very fancy infinitely large table!
Application reads information
Uses it to create a group of updates
Then uses group commit to install them atomically

Conflicts? One “wins” and the other “fails”, or perhaps both attempts fail
But this ensures that data moves in a predictable manner version by version: a form of the ACID model!

Thus BigTable offers strong consistency, up to a limit
There are failure cases they deliberately don’t cover
## Microbenchmarks

<table>
<thead>
<tr>
<th>Experiment</th>
<th># of Tablet Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>random reads</td>
<td>1212</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
</tr>
</tbody>
</table>
Performance

![Graph showing performance data]
# Application at Google

<table>
<thead>
<tr>
<th>Project name</th>
<th>Table size (TB)</th>
<th>Compression ratio</th>
<th># Cells (billions)</th>
<th># Column Families</th>
<th># Locality Groups</th>
<th>% in memory</th>
<th>Latency-sensitive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl</td>
<td>800</td>
<td>11%</td>
<td>1000</td>
<td>16</td>
<td>8</td>
<td>0%</td>
<td>No</td>
</tr>
<tr>
<td>Crawl</td>
<td>50</td>
<td>33%</td>
<td>200</td>
<td>2</td>
<td>2</td>
<td>0%</td>
<td>No</td>
</tr>
<tr>
<td>Google Analytics</td>
<td>20</td>
<td>29%</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0%</td>
<td>Yes</td>
</tr>
<tr>
<td>Google Analytics</td>
<td>200</td>
<td>14%</td>
<td>80</td>
<td>1</td>
<td>1</td>
<td>0%</td>
<td>Yes</td>
</tr>
<tr>
<td>Google Base</td>
<td>2</td>
<td>31%</td>
<td>10</td>
<td>29</td>
<td>3</td>
<td>15%</td>
<td>Yes</td>
</tr>
<tr>
<td>Google Earth</td>
<td>0.5</td>
<td>64%</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>33%</td>
<td>Yes</td>
</tr>
<tr>
<td>Google Earth</td>
<td>70</td>
<td>–</td>
<td>9</td>
<td>8</td>
<td>3</td>
<td>0%</td>
<td>No</td>
</tr>
<tr>
<td>Orkut</td>
<td>9</td>
<td>–</td>
<td>0.9</td>
<td>8</td>
<td>5</td>
<td>1%</td>
<td>Yes</td>
</tr>
<tr>
<td>Personalized Search</td>
<td>4</td>
<td>47%</td>
<td>6</td>
<td>93</td>
<td>11</td>
<td>5%</td>
<td>Yes</td>
</tr>
</tbody>
</table>
So where does this table live?

- Google actually stores the table inside its “global file system” or GFS

- GFS is a kind of DHT too!
  - It has a “name node” service where you can find a list of data nodes and a key-value mapping from file name to data nodes that host that file (actually, “chunks” of that file, because Google’s files are huge)
  - Sort of the same idea as we just saw for BigTable
GFS and Chubby

- **GFS file system used under the surface for storage**
  - Has a master and a set of chunk servers
  - To access a file, ask master... it directs you to some chunk server and provides a capability
  - That server sends you the data

- **Chubby lock server**
  - Implements locks with varying levels of durability
  - Implemented over Paxos, a protocol we’ll look at a few lectures from now
GFS Architecture

Client

Master

Metadata

Chunkservername

Linux FS

Chunkservername

Linux FS

(request for metadata)

(metadata response)

(read/write request)

(read/write response)
Write Algorithm is trickier

1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations. Data is stored in chunkservers’ internal buffers.
5. Client sends write command to primary.
Write Algorithm is trickier

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.

7. Primary sends serial order to the secondaries and tells them to perform the write.

8. Secondaries respond to the primary.

9. Primary responds back to client.

Note: If write fails at one of chunkservers, client is informed and retries the write.
Write Algorithm is trickier
Write Algorithm is trickier
We’ve only scratched the surface

- We’ve focused on scalable storage

- But there are many other major, important services

- What are some examples? We don’t have time for deep dives but can at least mention a few...
Zookeeper

- Created at Yahoo!
- Integrates locking and storage into a file system
  - Files play the role of locks
  - Also has a way to create unique version or sequence numbers
  - But basic API is just like a Linux file system
- Implemented using virtual synchrony protocols (we’ll study those too, when we talk about Paxos)
- Extremely popular, widely used
At Facebook the need is sort of different from Google or Yahoo

- Facebook is concerned mostly with the relationships between people and companies or other things
- A likes B (and B likes A), A friends B (and B friends A), A tracks updates by B (and B sends updates to A)...
- This involves a whole collection of “graphs”

- TAO is a fancy file system like GFS or Zookeeper but specialized to track these kinds of graphs
Some TAO ideas

- One is to try and use key-value caching “at the edge” to be super fast even at risk of using stale data for a little while
- Updates flow in a huge pipeline towards the core of the cloud, are done in a batched way, then cache-updates flow back out.
- Every layer has redundancy and backup options if the layer below it is broken temporarily

We’ll study TAO more carefully later…
What about MapReduce (Hadoop)

- So famous that people have heard of it as often as TCP/IP or XML...

- This is an example of a service that lives deeper in the cloud and isn’t used “while” processing requests from clients

- Instead, MapReduce is useful for “offline” tasks
MapReduce

- Used for functional style of computing with massive numbers of machines and huge data sets
- Works in a series of stages
  - **Map** takes some operations and “maps” it on a set of servers so that each does some part
  - The operations are functional: they don’t modify the data they read and can be reissued if needed
  - Result: a large number of partial results, each from running the function on some part of the data
  - **Reduce** combines these partial results to obtain a smaller set of result files (perhaps just one, perhaps a few)
- Often iterates with further map/reduce stages
Hadoop

- Open source MapReduce
  - Has many refinements and improvements
  - Widely popular and used even at Google!

- Challenges
  - Dealing with variable sets of worker nodes
  - Computation is functional; hard to accommodate adaptive events such as changing parameter values based on rate of convergence of a computation
Classic MapReduce examples

- Make a list of terms appearing in some set of web pages, counting the frequency
- Find common misspellings for a word
- Sort a very large data set via a partitioning merge sort

Nice features:
- Relatively easy to program
- Automates parallelism, failure handling, data management tasks
The database community dislikes MapReduce

- Databases can do the same things
- In fact can do far more things
- And database queries can be compiled automatically into MapReduce patterns; this is done in big parallel database products all the time!

Counter-argument:

- Easy to customize MapReduce for a new application
- Hadoop is free, parallel databases not so much…
We’ve touched upon a series of examples of cloud computing infrastructure components
- Each really could have had a whole lecture

They aren’t simple systems and many were very hard to implement!
- Hard to design… hard to build… hard to optimize for stable and high quality operation at scale
- Major teams and huge resource investments
- Design decisions that may sound simple often required very careful thought and much debate and experimentation!
Some recurring themes

- Data replication using (key,value) tuples
- Anticipated update rates, sizes, scalability drive design
- Use of multicast mechanisms: Paxos, virtual synchrony
- Need to plan adaptive behaviors if nodes come and go, or crash, while system is running
- High value for “latency tolerant” solutions
  - Extremely asynchronous structures
  - Parallel: work gets done “out there”

Many offer strong consistency guarantees, but not necessarily for every aspect. “Selective” guarantees.