Today

1. Scaling up: Reliable multicast

2. Scaling out: Partitioning, DHTs and Chord
Recall the tradeoffs

• CAP: Distributed systems can have 2 of the following 3:
  – Consistency (Strong/Linearizability)
  – Availability
  – Partition Tolerance: Liveness despite arbitrary failures

• Bit of an oversimplification. Really: When you get P, do you choose A or C?

• Goal? ALPS (Coined by Lloyd and Freedman, 2011)
  – Available
  – Low-Latency
  – Partition-Tolerant
  – Scalable (to more than 1 “CPU” per site)
Reliable multicast motivation: Fast content purging in a CDN

- [Fastly CDN]

- CDN servers geographically distributed containing many replicas of data

- Want to give customers the ability to takedown content in a short period of time from across the CDN

- Your sales team advertises 150 ms takedown latency
Reliable multicast protocols

- Reliable Multicast Transport Protocol (RMTP)
- Scalable Reliable Multicast (SRM)

- Sequenced, lossless bulk delivery of data from one sender to a group of receivers

- TCP-like cumulative sequence numbers on data
- Sequence numbers and bitmaps in acknowledgement packets back to sender
- Window-based flow control
- Retransmissions, failure monitoring among receivers
Reliable multicast performance

Average throughput on nonperturbed members

“Sleep” time (fraction) of ONE group member

- Group size: 32
- Group size: 64
- Group size: 96
Bimodal multicast

- **pbcast**: Probabilistic broadcast
  - Birman *et al.*, ACM ToCS 17(2) 1999

- Atomicity property is the **bimodal delivery guarantee**:
  - High probability that each multicast will reach **almost all** processes
  - Low probability that a multicast will reach just a **very small set** of processes
  - Vanishingly small probability that it will reach some **intermediate number of processes**

- The traditional “all or nothing” guarantee thus becomes “almost all or almost none.”
Bimodal multicast (1/4)

Initially use UDP/IP best effort multicast
Periodically (e.g. every 100ms):

- Each node picks a random group member, and send a digest describing its state.
Recipient checks digest against own history

For all missing message, solicits copy of msg
Nodes respond to solicitations by retransmitting requested message(s)
Why “bimodal?”

- Two phases? Nope…
- Description of dual “modes” of result

![Graph showing bimodal distribution of Pbc delivery]

- Either sender fails…
- … or data gets thru w.h.p.
Epidemic algorithms via gossiping

- Assume a fixed population of size $n$

- For simplicity, assume “epidemic” (delivery) spreads homogenously through popularly
  - Simple randomized delivery: any one can deliver to any one with equal probability

- Assume that $k$ members are already delivered

- Delivery occurs in rounds
Probability of delivery

- Probability $P_{\text{deliver}}(k,n)$ that a undelivered member is delivered to in a round, if $k$ are already infected?

$$P_{\text{deliver}}(k,n) = 1 - P(\text{nobody delivers}) = 1 - (1 - 1/n)^k$$

$$E[\# \text{ newly delivered}] = (n - k) \times P_{\text{deliver}}(k,n)$$

- Basically it’s a Binomial distribution

- # rounds to deliver to the entire population is $O(\log n)$
Two prevailing styles

- **Gossip pull** ("anti-entropy")
  - A asks B for something it is trying to "find"
  - Commonly used for management replicated data
    - Resolve differences between DBs by comparing digests

- **Gossip push** ("rumor mongering"):
  - A tells B something B doesn’t know
  - Gossip for multicasting
    - Keep sending for bounded period of time: $O(\log n)$
  - Also used to compute aggregates

- **Push-pull gossip**
  - Combines both mechanisms
  - $O(n \log \log n)$ msgs to spread rumor in $O(\log n)$ time
Wednesday reading: *Bayou*

- **Problem:** Collaborative applications
  - E.g., Meeting room scheduling, bibliographic database

- **Setting:**
  - Mobile computing environment
  - Seek to support disconnected workgroups
  - Rely only on weak/opportunistic connectivity (occasional, pair-wise communication)

- **Key technical problem:** How to converge to (eventually) consistent state?
  - Use anti-entropy for pair-wise resolution
  - Observation: Need application-specific conflict detection and resolution at granularity of individual updates
1. Scaling up: Reliable multicast

2. Scaling out: Partitioning, DHTs and Chord
Scaling out by partitioning data

- Every data object belongs to data “partition”

- Each partition resides on one or more nodes
  - Replication protocol between nodes hosting partition, e.g., could be strong or eventually consistent

- Every node hosts one or more partition
What is a DHT?

• Single-node hash table abstract:
  key = Hash(name)
  put(key, value)
  get(key) → value
  – Service: $O(1)$ storage

• How do I do this across millions of hosts on the Internet?
  – Distributed Hash Table
What Is a DHT? (and why?)

• Distributed Hash Table:
  \[
  \text{key} = \text{Hash(data)} \\
  \text{lookup(key)} \rightarrow \text{IP address} \quad \text{(Chord)} \\
  \text{send-RPC(IP address, PUT, key, value)} \\
  \text{send-RPC(IP address, GET, key)} \rightarrow \text{value}
  \]

• The first step towards truly large-scale distributed systems
  – a tuple in a global database engine
  – a data block in a global file system
  – rare.mp3 in a P2P file-sharing system
DHT Factoring

- Application may be distributed over many nodes
- DHT distributes data storage over many nodes
Why the put()/get() DHT interface?

• API supports a wide range of applications
  – DHT imposes no structure/meaning on keys

• Key/value pairs are persistent and global
  – Can store keys in other DHT values
  – And thus build complex data structures
Why might DHT design be hard?

- Decentralized: no central authority
- Scalable: low network traffic overhead
- Efficient: find items quickly (latency)
- Dynamic: nodes fail, new nodes join
- General-purpose: flexible naming
The Lookup problem

- At the heart of all DHTs
Motivation: Centralized lookup (Napster)

SetLoc(“title”, N4)
Publisher@ N4
Key=“title”
Value=file data…

N1 N2 N3
DB

N6 N7 N8

Client

Lookup(“title”)

Simple, but $O(N)$ state and a single point of failure
Motivation: Flooded Queries (Gnutella)

Robust, but worst case $O(N)$ messages per lookup
Motivation: FreeDB, Routed DHT Queries (Chord, &c.)

Publisher

Key = H(audio data)
Value = {artist, album title, track title}

Lookup \( H(\text{audio data}) \)

Client
DHT Applications

• They’re not just for stealing music anymore…
  – global file systems [OceanStore, CFS, PAST, Pastiche, UsenetDHT]
  – naming services [Chord-DNS, Twine, SFR]
  – DB query processing [PIER, Wisc]
  – Internet-scale data structures [PHT, Cone, SkipGraphs]
  – communication services [i3, MCAN, Bayeux]
  – event notification [Scribe, Herald]
  – File sharing [OverNet]
Basic Approaches

• Require two features:
  – Partition management:
    • On which node(s) to place a partition
    • Including how to recover from a node failure, e.g., bringing another node into partition group
    • Changes in system size, e.g., nodes joining and leaving
  – Resolution:
    • Maintain mapping from data name to responsible node(s)

• Centralized: Cluster manager
• Decentralized: Deterministic hashing and algorithms
The partitioning problem

• Consider problem of data partition:
  – Given document X, choose one of k servers to use

• Suppose we use modulo hashing
  – Number servers 1..k
  – Place X on server \( i = (X \mod k) \)
    • Problem? Data may not be uniformly distributed
  – Place X on server \( i = \text{hash}(X) \mod k \)
    • Problem? What happens if a server fails or joins \( (k \rightarrow k\pm1) \)?
    • Problem? What if different clients have different estimate of \( k \)?
    • Answer: All entries get remapped to new nodes!
Placing objects on servers

• How to determine the server on which a certain object resides?

• Typical approach: Hash the object’s identifier

• Hash function $h$ maps object id $x$ to a server id
  – E.g., $h(x) = [ax + b \pmod{p}]$, where

  • $p$ is a prime integer

  • $a$, $b$ are constant integers chosen uniformly at random from $[0, p - 1]$

  • $x$ is an object’s serial number
Difficulty: Changing number of servers

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

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

Adding a machine results in all objects’ assignments changing: 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

• Simple to analyze
Chord IDs

• 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 [Karger ‘97]

A key is stored at its successor: node with next-higher ID
Basic Lookup

“N90 has K80”

“Where is key 80?”

K80
N90
N105
N120
N10
N32
N60

36
Simple lookup algorithm

`Lookup(my-id, key-id)`

n = my successor

if my-id < n < key-id            // next hop
   call Lookup(key-id) on node n
else                              // done
   return n

• Correctness depends only on successors
“Finger Table” Allows $\log(N)$-time Lookups
Finger $i$ Points to Successor of $n + 2^{i-1}$
Lookup with Fingers

Lookup\( (\text{my-id, key-id}) \)

look in local finger table for
highest node \( n \): \( \text{my-id} < n < \text{key-id} \)
if \( n \) exists
   call \( \text{Lookup(key-id)} \) on node \( n \)  // next hop
else
   return my successor  // done
Lookups Take $O(\log N)$ Hops
Join Operation

- N50 joins the ring via N15
- N50: send join(50) to N15
- N44: returns N58
- N50 updates its successor to N58
Periodic Stabilize

- N50: periodic stabilize
  - Sends stabilize message to N58
- N50: send notify message to N58
  - Update pred=44

\[
\text{stabilize(node=50)} \quad \text{notify(node=50)}
\]

\[
\text{succ=58} \quad \text{succ.pred=44} \quad \text{succ=44} \quad \text{succ=4} \quad \text{succ=58} \quad \text{succ=58} \quad \text{succ=58} \quad \text{succ=58}
\]

\[
\text{pred=nil} \quad \text{pred=35} \quad \text{pred=44} \quad \text{pred=44} \quad \text{pred=44} \quad \text{pred=44} \quad \text{pred=44} \quad \text{pred=44}
\]
Periodic Stabilize

- N44: periodic stabilize
- N44 Asks N58 for pred → N50
- N44 updates successor to N50
Periodic Stabilize

- N44 has a new successor: N50
- N44 notifies N50

N44 notifies node 44:
- succ=50
- pred=nil

N50 notifies node 50:
- succ=4
- pred=50
Periodic Stabilize Converges!

- This completes the joining operation!
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
4. Set N25’s successor pointer

- Predecessor pointer allows link to new node
- Update finger pointers in the background
- Correct successors produce correct lookups
Failures Might Cause Incorrect Lookup

N80 doesn’t know correct successor, so incorrect lookup
Solution: Successor *Lists*

- Each node knows \( r \) immediate successors
- After failure, will know first live successor
- Correct successors guarantee correct lookups
- Guarantee is with some probability
Choosing Successor List Length

• Assume ½ the nodes fail

• \( P(\text{successor list all dead}) = \left(\frac{1}{2}\right)^r \)
  – *i.e.*, \( P(\text{this node breaks the Chord ring}) \)
  – Depends on independent failure

• Successor list of size \( r = O(\log N) \) makes this probability
  \( 1/N \): low for large \( N \)
Lookup with Fault Tolerance

Lookup(my-id, key-id)
  look in local finger table and successor-list
  for highest node n s.t. my-id < n < key-id
  if n exists
    call Lookup(key-id) on node n  // next hop
  if call failed,
    remove n from finger table or successor-list
  return Lookup(my-id, key-id)
else return my successor  // done
Experimental Overview

- Quick lookup in large systems
- Low variation in lookup costs
- Robust despite massive failure

Experiments confirm theoretical results
Chord Lookup Cost is $O(\log N)$

Constant is $1/2$
Failure Experimental Setup

- Start 1,000 CFS/Chord servers
  - Successor list has 20 entries
- Wait until they stabilize
- Insert 1,000 key/value pairs
  - Five replicas of each
- Stop X% of the servers
- Immediately perform 1,000 lookups
DHash Replicates Blocks at $r$ Successors

- Replicas are easy to find if successor fails
- Hashed node IDs ensure independent failure
Massive Failures Have Little Impact

(1/2)^6 is 1.6%
DHash summary

• Builds key/value storage on Chord

• Replicates blocks for availability
  – Stores k replicas at the k successor servers after the block’s successor on the Chord ring

• Caches blocks for load balance
  – Client sends copy of block to each of the servers it contacted along the lookup path

• Authenticates block contents
DHTs: A Retrospective

- Original DHTs (CAN, Chord, Kademlia, Pastry, Tapestry) proposed in 2001-02

- Following 5-6 years saw proliferation of DHT-based applications:
  - filesystems (e.g., CFS, Ivy, Pond, PAST)
  - naming systems (e.g., SFR, Beehive)
  - indirection/interposition systems (e.g., i3, DOA)
  - content distribution systems (e.g., Coral)
  - distributed databases (e.g., PIER)
What DHTs Got Right

- **Consistent hashing**
  - Elegant way to divide a workload across machines
  - Very useful in clusters: actively used today in Dynamo, FAWN-KV, ROAR, ...
- **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, control, monitor
  - …well suited to “illegal” applications, be they sharing music or resisting censorship
DHTs’ limitations

• High latency between peers

• Limited bandwidth between peers (as compared to within a cluster)

• Lack of trust in peers’ correct behavior
  – securing DHT routing hard, unsolved in practice
Next time

- Wednesday 10/28 Paper Discussion: Weakening Consistency
- Bayou, Dynamo, Eiger