

# Distributed Software Development

## More Distributed Hash Tables

Chris Brooks

Department of Computer Science  
University of San Francisco

# Distributed Hash Tables

- On Tuesday, we talked about Distributed Hash Tables
  - Used Chord as an example
- Able to act as a distributed storage and indexing mechanism
- Maps “keys” (hashes of data) to “values” (coordinates in a space of nodes.)
- Each node is responsible for a set of the keyspace.

# Desirable Properties

- Desirable properties of DHTs include:
  - Always able to find data stored in the network
  - Able to support entry and exit of nodes
  - Able to tolerate node failure
  - Efficient routing of queries

# Chord

- Recall that Chord uses a 1-D ring to structure the network.
- Nodes keep a *finger table* that tells them the successors at various points in the network.
- Routing requires  $O(\log n)$  messages.
- Can tolerate failures through replication of data.

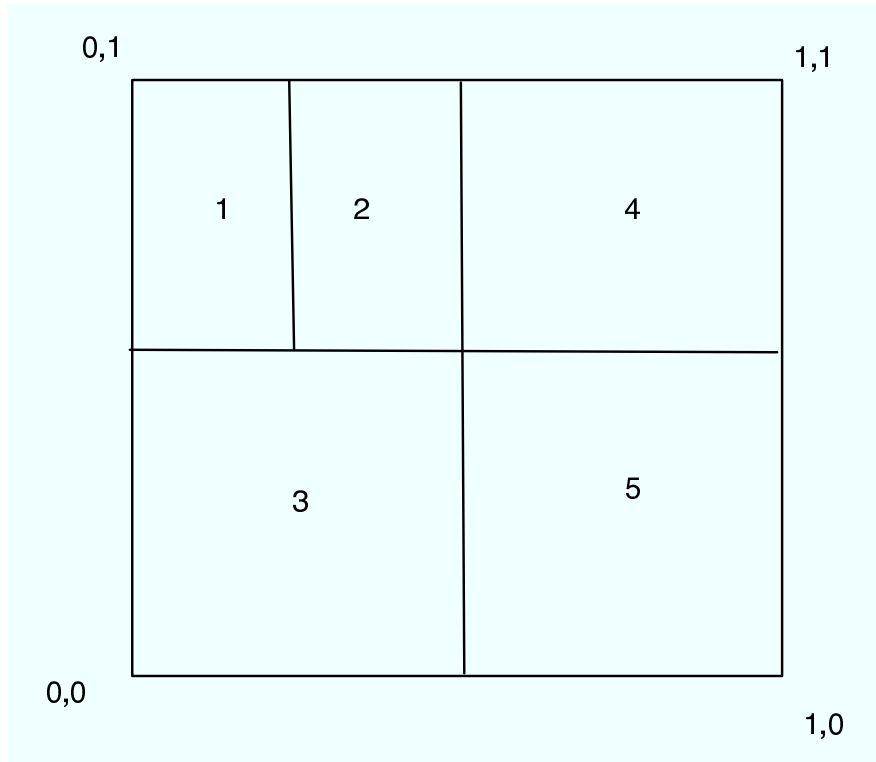
# CAN

- CAN stands for Content Addressable Network
- Developed at Berkeley at the same time as Chord was developed at MIT.
- Also provides hash-table like functionality.
  - get, store, delete

# Address construction in CAN

- CAN uses a  $d$ -dimensional torus as its address space.
  - Vector of  $d$  dimensions where each dimension “wraps around”
- Each node is responsible for a *zone* in this space.
- The hash function maps data into a point in this space.

# Example



- An example CAN in a 2-D space.
- Node 1 is responsible for data in the range  $( (0-0.25), (0.5-1) )$
- Node 4 is responsible for data in the range  $( (0.5-1), (0.5-1) )$

# Routing in CAN

- Each node keeps track of the IP address and zone of its neighbors.
- A neighbor is a node whose zone overlaps a node in  $n - 1$  dimensions, and abuts it in the  $n$ th dimension.
- In our example, 2,3, and 4 are neighbors of 1.
  - Remember that the space is really a torus.
- Routing is done by greedy search.
- Always forward the message to the neighbor who is closest to the data, using standard Euclidean distance. (ties broken randomly)

# Routing in CAN

- In a  $d$ -dimensional space of  $n$  evenly-sized zones,
  - Routing requires  $\frac{d}{4}(n^{\frac{1}{d}})$  hops.
  - Each node stores info about  $2d$  neighbors.
- By increasing  $d$ , we can reduce the length of routes
- Cost: additional storage at each node.
- We get resistance to failure when routing for free
  - If we are trying to route through a non-responsive node, just choose the “next-best” path.

# Node joining

- New nodes are added to the system by choosing an existing zone and subdividing it.
- First, the new node must find an existing bootstrap node.
- Then it must find its coordinates within CAN
- Finally, it must update its neighbors' routing tables.

# Node joining - bootstrapping

- It is assumed that the IP address of at least one current CAN node is known to the joining node.
- The designers use DNS to map a CAN hostname to one or more CAN nodes.
- This bootstrap node can provide the IP addresses of other nodes in the network that can be used as entry points.

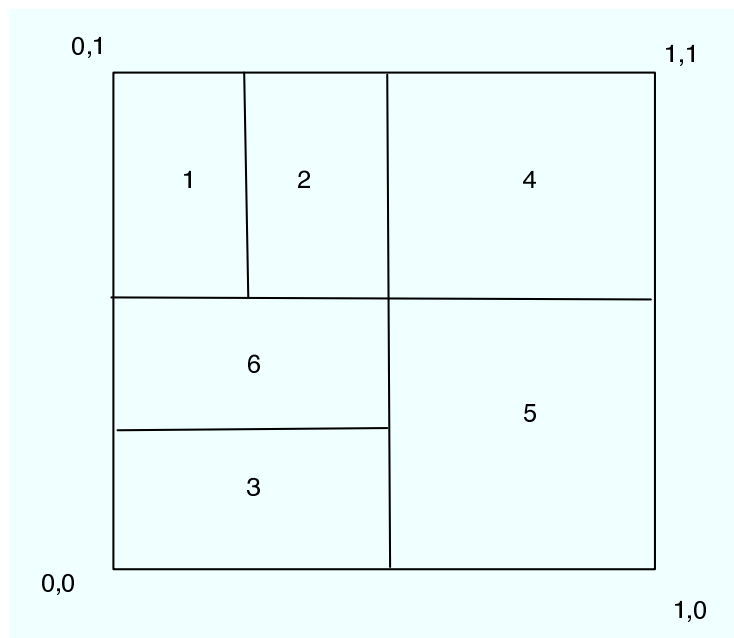
# Node joining - finding a zone to share

- The newly-joining node next randomly selects a point  $P$  in the coordinate space.
- It sends a JOIN message to that coordinate.
- This message is routed as all other CAN messages.
- When it reaches the node responsible for  $P$ , that node returns a reply to the newly-joining node.
- The node responsible for  $P$  then divides its zone in half and assigns half to the newly-joining node.

# Node joining - updating neighbors

- Once the zone has been partitioned, all neighbors need to update their routing tables.
- New node gets the routing table from the node whose zone it split.
- Adds node whose zone it split, and removes all non-neighbors.
- Split node removes non-neighbors as well.
- Both nodes send update messages to all affected nodes.

# Example



- Initially, 3's neighbors are 1,2 and 5.
- 6 joins and splits 3's zone along the x axis.
- 6 gets 3's list of neighbors
- 6 adds 3 to its neighbor list and sends an update to 1,2, and 5
- 3 adds 6 to its list of neighbors
- 3 removes 1 and 2 from its list of neighbors and sends them an update.

# Clean exit

- When a node decides to leave a CAN, it must give up its zone to one of its neighbors.
- If there is a neighbor who can be merged to form a valid CAN zone, this is done.
- Routing tables are updated as for a join.
- If not, the zone is temporarily given to the smallest neighbor, but not merged.
- Neighbor handles both zones separately.

# Handling Failure

- CAN nodes send periodic “I’m alive” messages to their neighbors
  - Contains a node’s zone coordinates and their neighbors and their coordinates.
- If a node does not receive an “I’m alive” message after a given time period, it begins a takeover.

# Handling Failure

- To begin, it starts a timer
  - Length is proportionate to its zone size.
  - Intuition: we want to merge smaller zones
- Once the timer expires, send a TAKEOVER message to all of the failed node's neighbors.
- When a TAKEOVER message is received, a node turns off its own timer if the zone volume of the sender is less than its own.
- Otherwise, it replies with its own TAKEOVER message.
  - Somewhat like leader election.

# Latency

- A weakness of CAN (and Chord) is that message routing happens at the *application level* rather than at the IP level.
- No guarantees that nodes near each other in the CAN network are actually near each other in the underlying network.
- This can lead to long delays in practice, even if the number of application-level hops is small
- Goal: achieve latency close to that of the underlying IP latencies.

# Improvements: multiple coordinate spaces

- One improvement is to maintain multiple coordinate spaces.
- Each node is assigned a different zone in each coordinate space, or “reality”.
- Contents are replicated for each reality.
- If there are  $r$  realities, this means that  $r$  copies of the data are stored in the network.
- This provides fault tolerance.
- By translating routing messages between realities, nodes can potentially find shorter-latency routes between nodes.

# Improved routing metrics

- The basic CAN routing algorithm uses greedy search. It chooses the node that produces the greatest reduction in distance between the present node and the data to be found.
- An alternative is to also measure the round-trip-time between a node and each of its neighbors.
- When choosing where to forward a packet, select the node that maximizes progress over RTT.
- This favors low-latency paths at the IP level.

# Overloading zones

- The basic design assumes that each zone is assigned to exactly one node.
- Alternatively, multiple peers can share zones.
- Each peer maintains a list of other peers in its zone, in addition to the neighbor list.
- When a node receives an update from a neighbor, he computes the RTT for each of the nodes in that zone.
- Retains the lowest RTT.
- Contents may be either divided amongst peers or else replicated.

# CAN summary

- Like Chord, CAN provides an implementation of a DHT.
- Uses a  $d$ -dimensional space to divide data.
- Space is broken and merged as nodes enter and leave.
- Explicitly trades off storage for routing efficiency.
- But Chord and CAN are active research projects (at MIT and Berkeley, respectively.)

# Coral

- Coral is a peer-to-peer Web content caching and distribution system.
- Designed to distribute the load of high-demand Web content.
- Also uses a form of DHTs
- Run by NYU
- To access “Coralized” content, append `’.nyud.net:8080’` to a URL.

# Coral in a Nutshell, pt 1

- Client sends a DNS request for foobar.com.nyud.net to its local resolver.
- DNS request is passed along to a Coral DNS server in the .net domain.
- The Coral DNS server probes the client to discover RTT, and the location of the last few network hops.
- Checks Coral to see if any HTTP proxies are near the client.
- DNS returns proxies (or sometimes nameservers) close to the client.

# Coral in a Nutshell, pt 2

- The client then sends an HTTP get to the specified proxy.
- If the proxy has a cache of the file, this is returned.
- Otherwise, the proxy looks up the object in Coral
- Object is either returned from another Coral node, or else from the original source.
- Coral proxy stores a reference to this item in Coral, indicating that the proxy now has a copy.

# Coral DNS

- When a lookup is done in DNS for a Coral address (anything ending with nyucd.net), one of the Coral DNS servers responsible for this domain tries to discover a Coral cache near the client.
- This client's IP address will be returned as a result of the DNS lookup.
- RTT and traceroute-style information is used to try to determine the latency to the client.

# Coral HTTP proxy

- The Coral proxies are responsible for cacheing and serving up content.
- Goal: fetch content from other clients whenever possible
  - The whole idea is to ease the burden on the original host.
- Each proxy keeps a local cache.
- On request, does the proxy have the data?
  - Yes. Send the data to the client. Done.
  - No, but a proxy we know does. Fetch the data and return it to the client.
  - No, and no nearby proxies have the data. Fetch the data from the origin.

# Coral Architecture

- Keys are hashed 160-bit identifiers (for data), or else hashed IP addresses (for nodes).
- Like Coral and CAN, this key is used to place the data in the system.
- This ID is then used to perform routing.

# Routing and DSHTs

- Coral uses a variant of the DHT called a distributed sloppy hash table to handle routing lookups.
- Each node keeps a hash table that maps keys to nodes that are *close to* the node storing the data.
  - In other words, they might hash to a node that's not quite right.
- When a piece of data is requested, a node looks up the key in its hash table and either finds:
  - The node storing the data
  - A node that is closer to the data (in keyspace terms)
    - XOR of the keys is used to determine distance.
  - Like Chord, Coral can find the actual node storing the data in a logarithmic number of hops.

# Sloppy Storage

- A potential problem for systems like Coral is the saturation of a few nodes hosting frequently-requested pieces of data.
  - This is sometimes called hot-spot congestion
- Coral solves this by using sloppy storage.
- Data is often stored at a node close to the hash value, rather than at the hash value itself.

# Sloppy Storage

- When a node wants to insert a key/value pair in Coral, it uses a two-phase technique.
- In the forward phase, it searches forward to the node with the hash value matching to the data to be stored.
- It stops whenever it finds a node that is both *full* and *loaded* for the key of the data to be stored.
  - *full*: The node stores enough values for the key that are sufficiently stale.
  - *Loaded*: The node has received sufficient requests for the key within a given time period.
- If a full and loaded node is discovered, the data is stored there and the algorithm stops.
- Otherwise, the node is placed on a stack and routing continues.

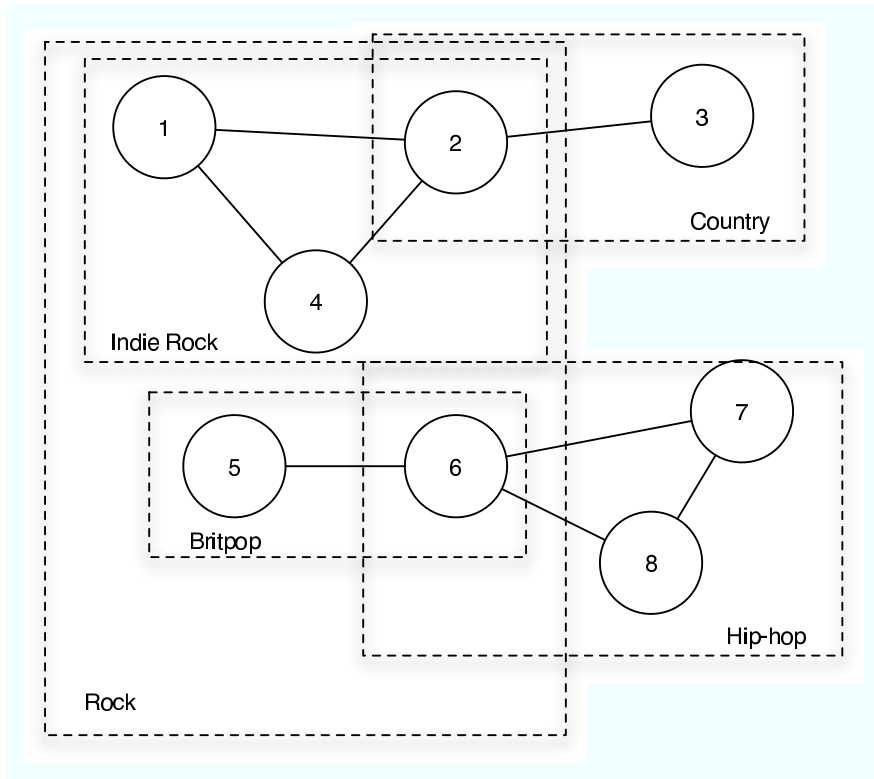
# Sloppy Storage

- In the second, or reverse, phase, the client pops previously-visited nodes off of the stack and determines whether they can store the data.
  - Do they have any old cache data that can be displaced?
- This serves to replicate the data at multiple nodes radiating out from its “correct” location.
- The more requests for the data, the farther away it will be written.
- This helps to avoid hot-spot congestion.
- Retrieval can happen as normal.

# Semantic Overlay Networks

- Three criticisms of DHT-style P2P networks are:
  - Forces nodes to store data and accept arbitrary connections from other peers.
  - Lookup is based on a hash function that has no relation to data content.
  - Aren't very useful for broader sorts of queries: approximate, range, logical.
- An alternative mechanism for improving routing in P2P networks is the construction of a *semantic overlay network*
  - This is work done by the P2P group at Stanford
- Idea: nodes in a network are connected to other nodes with similar content.

# Example



- Nodes are connected to other nodes that contain a significant amount of similar content.
- This creates a semantic overlay.
- A node might belong to multiple SONs, depending on its interests.
- Queries for content are forwarded only within the SON.

# Classifying Nodes

- A primary challenge in building SONs is classifying nodes (and queries).
  - What does it mean to say a node is a “Rock” node?
- The Stanford group constructs a number of hierarchical classification schemes based on data from [allmusic.com](http://allmusic.com)
  - Style/genre
  - Era
  - Mood
- A node is classified into one or more leaf concepts within this hierarchy.
  - Classification may be imprecise, in which case non-leaf nodes are used.

# Deciding which SONs to join

- How does a node decide which SONs to join?
  - conservative - join any SON for which you are eligible.
    - Data can always be found, but search is less efficient.
  - Aggressive - only join a subset of eligible SONs
    - More efficient, but risks not finding all data.
  - Experimentally, a “likelihood of discovery” parameter can be used to determine how many SONs a node should join.
- Once SONs are constructed, networks with a small number of members can be merged, and networks with a large membership can be further subdivided.

# Querying in a SON

- Queries require a service that can map a query onto the appropriate SON and then locate a node in the SON.
  - Each node can use the same classifier
  - Routing information can be propagated as in a DHT.
- The flooding query approach is used, but only within the SON, rather than within the whole network.

# Discussion

- SONs provide a middle ground between DHTs and unstructured P2P networks.
- Allow nodes to choose what to store and who to connect to.
- Require a mechanism for classifying nodes according to content.
  - Most effective with files for which there is an external hierarchy, such as allmusic.
- Less theoretically sound than DHTs
  - Most results are purely empirical.

# Summary

- DHTs provide a structured way to store and index information using a P2P model.
- Chord uses a logical ring, CAN uses a multi-dimensional torus.
- Both use greedy search to locate data.
  - Can provide efficient lookup as the network grows.
- Coral applies DHT technology to web cacheing and replication.
  - Uses sloppy DHTs to deal with congestion issues.
- Semantic Overlay Networks are an alternative approach to solving the routing problem in P2P networks.
  - Connect nodes with related content, and search only within those networks.