

Distributed Software Development

Distributed Hash Tables

Chris Brooks

Department of Computer Science
University of San Francisco

Network Characteristics

- What are some qualities of unstructured P2P networks?

Network Characteristics

- What are some qualities of unstructured P2P networks?
- Peers can connect to any other peer
- Peers can choose what data to store
- No fixed peer addressing scheme
- Peers can easily come and go

Network Characteristics

- What are the implications of this?

Network Characteristics

- What are the implications of this?
- Search may be slow or incomplete
- Data may not always be available
- Data may be difficult to locate
- It may be difficult to find a particular peer

Applications

- What sorts of applications are a good fit for this sort of network?
- What sorts of applications are a poor fit?

Structured P2P networks

- So far, we've focused on P2P apps in which users may join the network at any point, and may store whatever data they want.
 - Network has no particular topology.
 - No guarantees that all nodes in the network can reach each other.
- These are referred to as *unstructured* P2P networks.
- We can also talk about *structured* P2P networks
 - Topology, number of connections, file placement is fixed.

Distributed Hash Tables

- The most common structured P2P application is the *distributed hash table*
 - We can build a file system or storage scheme on top of this.
- Supports a single operation: map keys to node IDs.
- Key is a unique data identifier.

DHT - design goals

- We would like for DHTs to have the following properties:
 - Data can always be found if it is in the network.
 - Each node only has to know about a small number of other nodes.
 - Number of messages needed to find data is small
 - Can tolerate crash failure
 - Can handle nodes entering and leaving

Hash tables - review

- Hash tables provide a mapping from keys to values.
- They do this by means of a *hash function*
 - For example, modulo can be used as a hash function
- Good hash functions distribute hashed values evenly.
- Problem: What if we add buckets to our hash table?
 - Might need to re-hash all our data!
- How can we avoid this?

Consistent Hashing

- Consistent hashing is a particular type of hash function.
- It has the property that when the number of buckets (or nodes in the network) is changed, at most $O(\lg N)$ items will need to be re-hashed.
- This means that, in a distributed hash table, we can efficiently add or subtract nodes without large-scale updating.

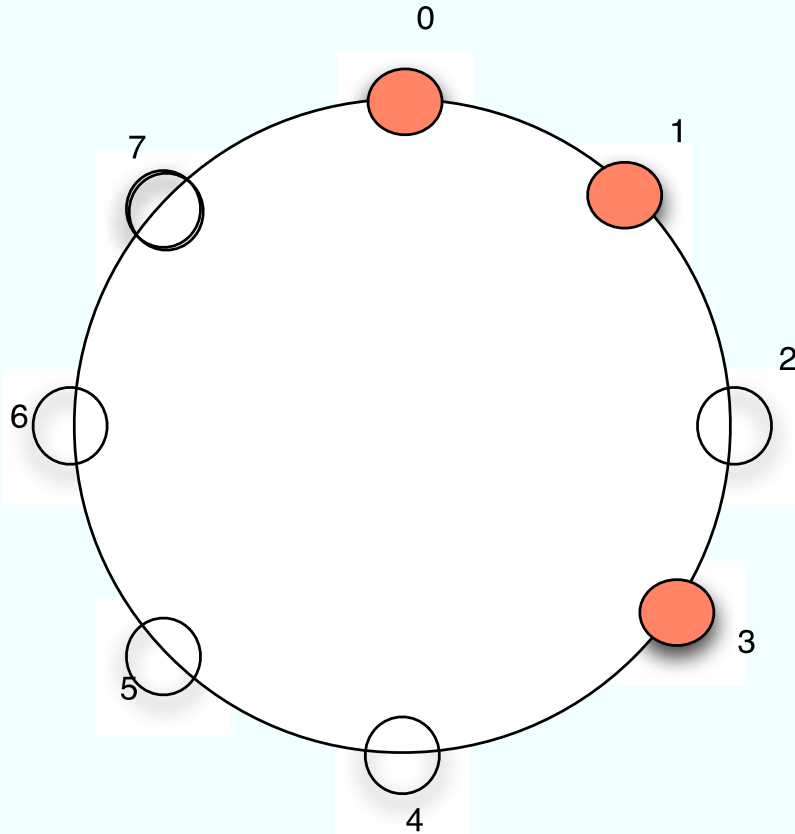
Chord

- Our discussion of DHTs will focus on Chord.
 - Developed at MIT
 - One of four DHT schemes (Can, Pastry, Tapestry are the others) developed contemporaneously.
- Designed to serve as the basis of a cooperative file system.

Chord nodes

- Each Chord node begins with an m bit identifier obtained by hashing its IP address.
- Nodes are arranged in a circle modulo 2^m .
- Data is hashed with the same function (SHA-1) to produce a *key*
- A node is responsible for storing all data whose key is between its identifier and its successor's identifier.

Example



- Three nodes in this example, with identifiers 0,1,3.
- 0 is responsible for data that hashes to 0.
- 1 is responsible for data that hashes to 1 or 2.
- 3 is responsible for data that hashes to 3-7.

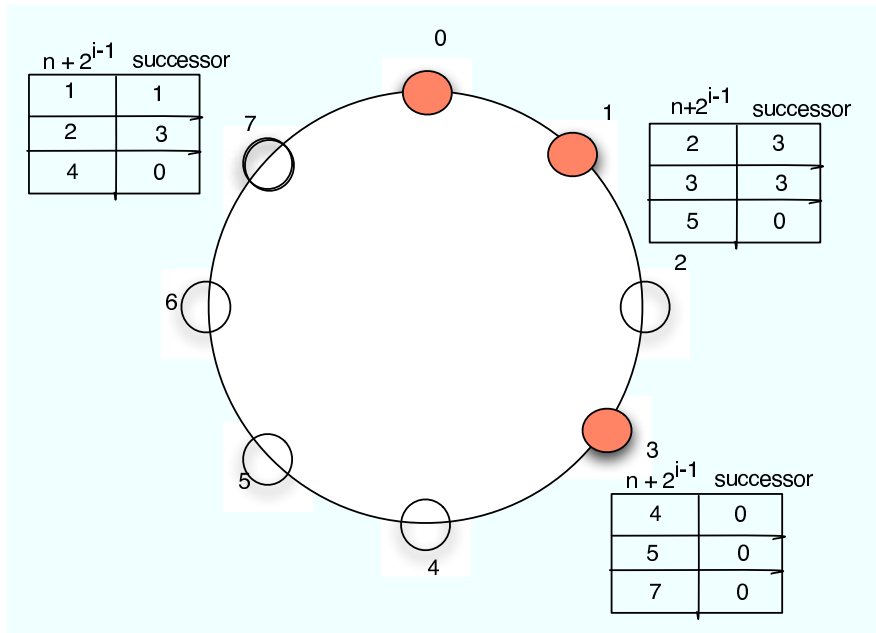
Simple Routing

- The simplest way to routing in Chord is as follows:
 - Each node keeps track of one other node: its successor.
 - To find a piece of data with key k , if a node does not have k , it asks its successor.
 - Eventually, all nodes in the ring are queried.
- Very simple, but requires N messages.

Actual Chord Routing

- What Chord really does is have each node keep a *finger table*
- m entries in the table.
- For node n , the i th entry is the first node that succeeds n by 2^{i-1} modulo n .
- Contains the Chord identifier and the actual IP address of the node.

Example



- In this case, each finger table has three entries.
- Indexes the nodes storing data 1, 2 and 4 keys away.
- Each node only needs to contain $O(\log N)$ entries.

Routing

- To find a piece of data, a node computes the data's hash, yielding its key.
- Three possibilities:
 - The node is storing the data itself.
 - The key is in the seeking node's finger table.
 - The key is not in the seeking node's finger table.
- If the key is not in the seeking node's finger table, we execute find-predecessor.

Routing

- Find-predecessor is meant to find the node that is before the data we are trying to find.
 - In the example, the predecessor of 5 is 3.
- Each node sends a message to the node in its finger table who is closest to the data being sought without going over.
- If a message is received by a node n to find data with key k , and $successor(n) > k$, then n has the data.
- Otherwise, call find-predecessor on the successor on n .
- Example: At node 3, want to find 2.
- We know that $successor(7)$ is 0. We then query 0 to find that 2 is stored by node 1.

Routing

- Chord is guaranteed to find data in $O(\log N)$ messages.
 - Assuming that SHA-1 is a fair hashing function.
- In this way, the network can grow to large numbers of nodes while still retaining fast search.
- How is this different from the way Gnutella handles network structure and routing?

Node Joining

- One of the hallmarks of P2P networks is the ability for nodes to join and leave dynamically.
- Goals:
 - No data is lost. This requires:
 - Each node's successor is correctly maintained.
 - For each key k , $successor(k)$ is responsible for k .
 - For the network to be efficient, finger tables must also be updated.

Node Joining

- To simplify things, let's assume that every node knows the node before it (its predecessor) in the ring.
 - (It can find this by searching for the next smallest key with $O(\log N)$ messages)
- When a new node n^* joins the network, the following things must happen:
 - Its predecessor pointer and finger table are initialized.
 - Other nodes' predecessor pointers and finger tables are updated as necessary.

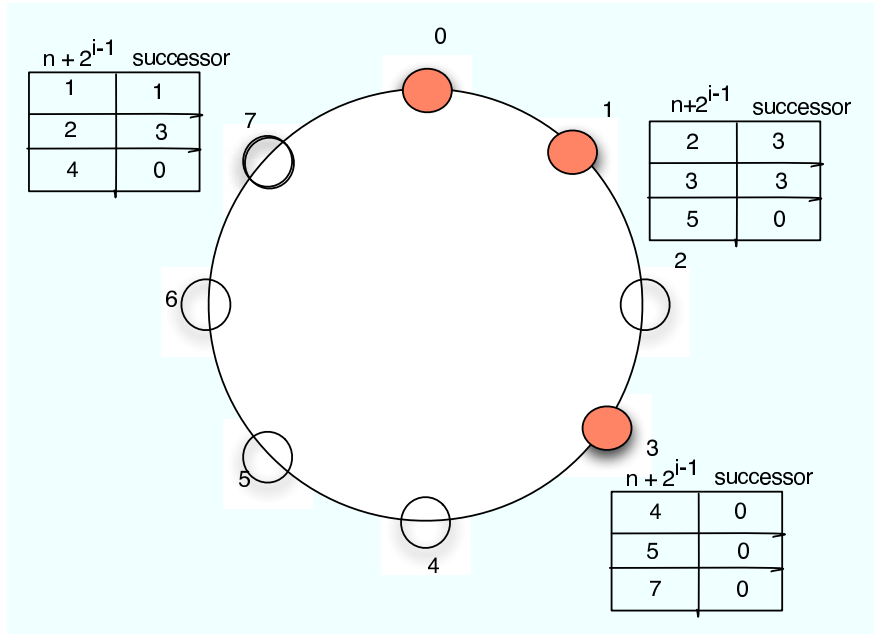
Node Joining

- When node n^* wants to join the network, it contacts some other node j that it already knows about.
- It asks j to find its predecessor and fill in its finger table by looking up each of the values.
- As a practical optimization, start with the finger table of n^* 's immediate successor.

Node joining

- We may also need to update the finger tables of other nodes.
 - Example: If we added a node 6 in our example, we would need to update the finger tables of nodes 2 and 3.
- For each node a in the network, our new node n^* will become the i th finger of a if:
 - $a + 2^{i-1} < n^*$
 - the i th finger of a succeeds n^*
- Start with the predecessor of n^* and works “backwards”

Example



- for $i = 1$ to 3 :
 - find the predecessor of $n^* - 2^{i-1}$
 - update finger table - recurse if necessary.
- Add 6.
 - Update entry 1 of 3.
 - Update entry 2 of 3.
 - Update entry 3 of 0 and 1.

Node Joining

- Updating all finger tables takes $O(\log^2 n)$ messages.
- Lastly, all data for which node n^* is the successor must be transferred to n^* .
 - What this means may depend on the application.
- Note that this means that nodes do not have complete control over what they store.

Stabilizing

- While the network is being updated, other nodes may also be joining or issuing queries.
- The basic Chord algorithm may fail if a query is done before successors are updated or keys are transferred.
- This can be corrected by occasionally having each node run a stabilize algorithm.

Stabilizing

- On joining:
 - Find your successor and stop.
- Periodically, each node runs stabilize:
 - Find successor.
 - Find successor's predecessor - should we be that node's successor?
 - If so, update its finger table.
- This will allow the network to survive simultaneous joins.

Failure

- To deal with failure, have each node keep a list of its r first successors.
- If you notice your successor has failed, replace it with the first live successor in the list.
- We can also use this to replicate data by storing a copy of each key at each of the r successors.

Discussion

- DHTs provide a way to store and retrieve data in a P2P setting.
- Assumptions are different than in unstructured file-sharing networks:
 - Node may only join in specific places in the network
 - Nodes agree to allow foreign data to be stored on them.
 - All nodes have the same functionality and requirements.

Summary

- DHTs provide a P2P-like abstraction
- Chord uses a 1D keyspace to connect peers in a logical ring.
- Finger tables provide efficient routing.
- Can tolerate dynamic join/leave and failure.

Next Time

- CAN vs Chord
- Coral
- Overlay networks - routing based on content.