Distributed File Systems: An Overview of Peer-to-Peer Architectures

Distributed File Systems

- Data is distributed among many sources
  - Ex. Distributed database systems
  - Frequently utilize a centralized lookup server for addressing
- Completely distributed approach
  - No centralized services or information

Peer-to-peer Systems

- What is a P2P System?
  - Network of nodes with equivalent capabilities and responsibilities
- Common Examples
  - Gnutella
  - Freenet

Peer-to-Peer Systems

- P2P is a convenient paradigm which can be used to serve various applications (including the popular use of file sharing).
- Problem with P2P systems: locating the particular node which stores the requested data

Node Location: Napster

- Napster uses a central index to search for nodes with desired file.
- If central server goes down, service is lost for all users.
- NOT true P2P.

Node Location: Gnutella

- Search Algorithm: Random
- Gnutella broadcasts file requests to nodes
- Solution scales extremely poorly, therefore requests cannot be broadcast to all nodes.
- With Gnutella, search may fail even though the file exists in the system

Node Location: Freenet

- Search Algorithm: Random
- In the Freenet system, no particular node is responsible for a file. Instead searches look for cached copies of the file.
- Problems with Freenet: once again existing files are not guaranteed to be retrieved; no bound on cost.

Problems

- Random Search Algorithms:
  - Work poorly for uncommon data
  - No theoretical bound on the overhead required
- How to introduce determinism without centralized mechanisms?
  - Ans: Hash functions

Hash Functions

- A hash function is (usually) a one way function which will produce the same "key" given the same input
- Hash functions we consider will have follow a uniform distribution in keyspace

Hash Based P2P Systems

- Chord (Pastry is very similar)
  - Utilizes a ring based overlay
- CAN
  - Utilizes a hyper-space overlay model

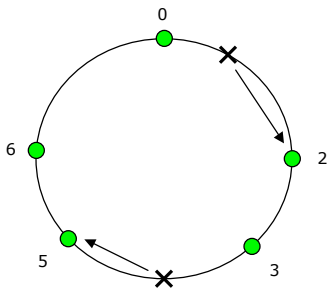
What Chord Can Contribute

- Chord is truly distributed: No node is more important than another.
- If requested object exists in the system, it will definitely be found.
- Chord gives performance bounds.

Consistent Hashing

- Using a consistent hashing function (SHA-1 function), Chord maps a given key to a particular node.
- Requests for object with a particular key are easily forwarded to the correct node.
- Consistent hashing helps to provide natural load balancing.

Mapping of Objects



- Node identifiers are established in a circle. Keys are assigned to the node with the next highest value.

Routing

- If every node has knowledge of its successor node, requests can be propagated along the ring.
- Problem: it may require traversing all N nodes in order to find the object.
- Solution: use *finger tables* to optimize routing.

Finger Tables

- Given *m* bits in the key/node identifiers, every node keeps a routing table with *m* entries (entries hold node identifier, IP address, and port numbers).
- The *i*th entry in the table at node *n* contains the identity of the node that succeeds *n* by at least 2^{i-1}.
- If *s* is the *i*th finger of the node, then *s*=successor(n+ 2^{i-1}), and is denoted by *n*.*finger*[*i*].*node*.

Finger Tables (cont.)

- The first entry of the finger table is the successor of *n*.
- Subsequent entries are spaced out more and more.
- If a node doesn't know the successor of a key *k*, it passes the request on to a node whose ID is closer. Thus the request is passed at least half the distance to the responsible node.

Finding Successors

- In order to find the successor of a key, the predecessor of the key is found, and the successor of that node is taken from its finger table.

```
n.find_successor(key)
  n'=find_predecessor(key);
  return n'.successor;
```

Finding Predecessors (1)

- In order to find the predecessor of a key, request is passed along to next closest node until key falls within appropriate interval.

```
n.find_predecessor(key)
  n'=n;
  while(id ∉(n',n'.successor])
   n'=n'.closest_preceding_finger(key);
  return n';
```

Finding Predecessors (2)

```
n.closest_preceding_finger(key)
  for i=m downto 1
    if (finger[i].node ∈(n, key))
      return finger[i].node;
  return n;
```

Performance

- Since the distance to the successor of a key is halved with each step, the bound for number of steps required for lookups is O(log N).

Node Joins: Invariants

- Every node maintains the correct successor.
- For every key k, node successor(k) is responsible for k.

Node Joins: Process

- Initialize the predecessor and fingers of new node *n* (to simplify joins and leaves, every node is also responsible for keeping a pointer to their predecessor).
- Update the fingers and predecessors of existing nodes.
- Notify the higher layer software so that state of keys is transferred appropriately (note that *n* is the only node to which any transfers occur).

Concurrent Operations

- Aggressively maintaining finger tables of all nodes difficult to maintain with concurrent joins.
- When a single node joins, very few finger table entries need to be modified.
- To adjust for concurrent joins, use a stabilization protocol instead of the aggressive correctness protocol.

Stabilization Pseudocode

- Node n knows of some other node n' when joining.

```
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);
```

Pseudocode (cont.)

- Successor's are verified periodically.

```
n.stabilize()
  x=successor.predecessor;
  if(x ∈ (n, successor))
    successor = x;
  successor.notify(n);
n.notify(n')
  if(predecessor is nil or n' ∈ (predecessor,n))
    predecessor = n';
```

Pseudocode (cont.)

- Finger tables are refreshed periodically.

```
n.fix_fingers()
  i=random index > 1 into finger[];
  finger[i].node =
            find_successor(finger[i].start);
```

Node Failures

- In order for failure recovery, queries must still succeed until system stabilizes.
- In order for successful queries, nodes must have correct knowledge of their successors.
- Every node keeps a list of its *r* nearest successors.

Quick Note on Pastry

- The Chord and Pastry designs are similar for the most part. The notable difference between the two is that Pastry provides locality while Chord does not.

CAN

- Use an *d*-dimensional hyperspace to distribute files
- A node is assigned to a point in the hyperspace using *d* hash functions
- The hyperspace is divided into partitions according to node placement (should be uniformly distributed due to hash functions)

CAN