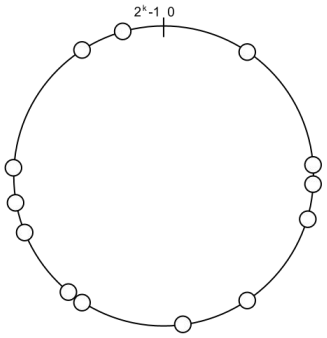
Network Centric Computing (CS-382)

Assignment 2



DHT Based Storage System

Intro:

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. To solve this problem, Chord a distributed lookup protocol can be used. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Behind this simple interface, Chord distributes objects over a dynamic network of nodes, and implements a protocol for finding these objects once they have been placed in the network. Every node in this network is a server capable of looking up keys for client applications, but also participates as key store. Hence, Chord is a decentralized system in which no particular node is necessarily a performance bottleneck or a single point of failure (if the system is implemented to be fault-tolerant).

Keys:

Every key (based on the name of a file) inserted into the DHT must be hashed to fit into the keyspace supported by the particular implementation of Chord. The hashed value of the key will take the form of an m bit unsigned integer. Thus, the keyspace (the range of possible hashes) for the DHT resides between 0 and 2^m-1 inclusive.

The Ring:

Just as each key that is inserted into the DHT has hash value, each node in the system also has a hash value in the keyspace of the DHT. To get this hash value, we could simply give each node a distinct name (or use the combination of IP and port) and then take the hash of the name, using the same hashing

algorithm we use to hash keys inserted into the DHT. Once each node has a hash value, we are able to give the nodes an ordering based on those hashes. Chord orders the node in a circular fashion, in which each node's successor is the node with the next highest hash. The node with the largest hash, however, has the node with the smallest hash as its successor. It is easy to imagine the nodes placed in a ring, where each node's successor is the node after it when following a clockwise rotation.

To locate the node at which a particular key-value pair is stored, one need only find the successor to the hash value of the key.

The Overlay:

As the paper in which Chord was introduced states, it would be possible to look up any particular key by sending an iterative request around the ring. Each node would determine whether its successor is the owner of the key, and if so perform the request at its successor. Otherwise, the node asks its successor to find the successor of the key and the same process is repeated. Unfortunately, this method of finding successors would be incredibly slow on a large network of nodes. For this reason, Chord employs a clever overlay network that, when the topology of the network is stable, routes requests to the successor of a particular key in log(**n**) time, where **n** is the number of nodes in the network.

This optimized search for successors is made possible by maintaining a finger table at each node. The number of entries in the finger table is equal to **m**, where **m** is the number of bits representing a hash in the keyspace of the DHT. Entry **i** in the table, with 0 <= **i** < **m**, is the node which the owner of the table believes is the successor for the hash h+2^i (**h** is the current node's hash). When node A services a request to find the successor of the key k, it first determines whether its own successor is the owner of k (the successor is simply entry 0 in the finger table). If it is, then A returns its successor in response to the request. Otherwise, node A finds node B in its finger table such that B has the largest hash smaller than the hash k, and forwards the request to B.

The Problem:

You will be implementing a simple version of Chord. Your implementation of Chord should support dynamic insertion and removal of nodes, and continue to serve get and put requests simultaneously

and correctly (if a value exists for a key, it must always be accessible). Keys should never reside at more than two nodes at any given time, and only one node when the ring is in a stable state.

Basic functions to implement should be:

Join(node)

Leave()

Find_successor(key) #summarized in Overlay

Get(key)

Put(key, value)

Joining (30):

Creating the ring is easy. The first node fills its finger table with references to itself, and has no predecessor. Then, when node A joins the network, it asks an existing node in the ring to find the successor of the hash of A. The node returned from that request becomes the successor of A. The predecessor of A is undefined until some other node notifies A that it believes that A is its successor.

Leaving (15):

In case a node wants to leave the chord, it should properly inform its predecessor, and transfer all the files to its successor before going offline.

Failure Resilience (15):

Every node should periodically contact its successor to know if it's still online, and in case of three non-replies assume that it's no longer online and update its successor. This is achieved by maintaining successor lists, rather than a single successor so that the failure of a few nodes is not enough to send the system into disrepair.

Downloading (20):

Client should be able to connect with any node and enquire about a file by its name. The node will compute the key of the file. If the node does not have the file itself it will contact its successor or contacting the node in its finger table that has the largest hash smaller than the key, and forwards the request to it. (more details in Overlay)

Storing (20):

Storing is done similarly to downloading. Given a file a key is computed based on the file name and the request is either kept locally or is forwarded to the node with the largest hash smaller than the key.

Important Note:

You can do this assignment in pairs. You can either write code in python or java.

If there are questions ask the course staff; they are there to help you. Other than your partner you are not allowed to discuss your code with anyone. Vivas will be individual and those who will not give the viva will get no marks for the assignment.

Also DO NOT plagiarize as any such act will be reported to the DC.

Submission Instructions:

- Submit all your code in a zip file with the roll number of both your teammates e.g. 18100206_19100206.zip
- Also submit a txt file explaining your approach and implementation. Also add individual partners' contribution in the assignment.
- The deadline for the assignment is 16th of March.