Yale CS 423 Final Project DHT

Mengci Liu, Chen Xu
Net ID: ml862, cx44

1. **Features:**

   (a) Chord strcture maintainance by maintaining neighbours.

   (b) Key value pair insertion.

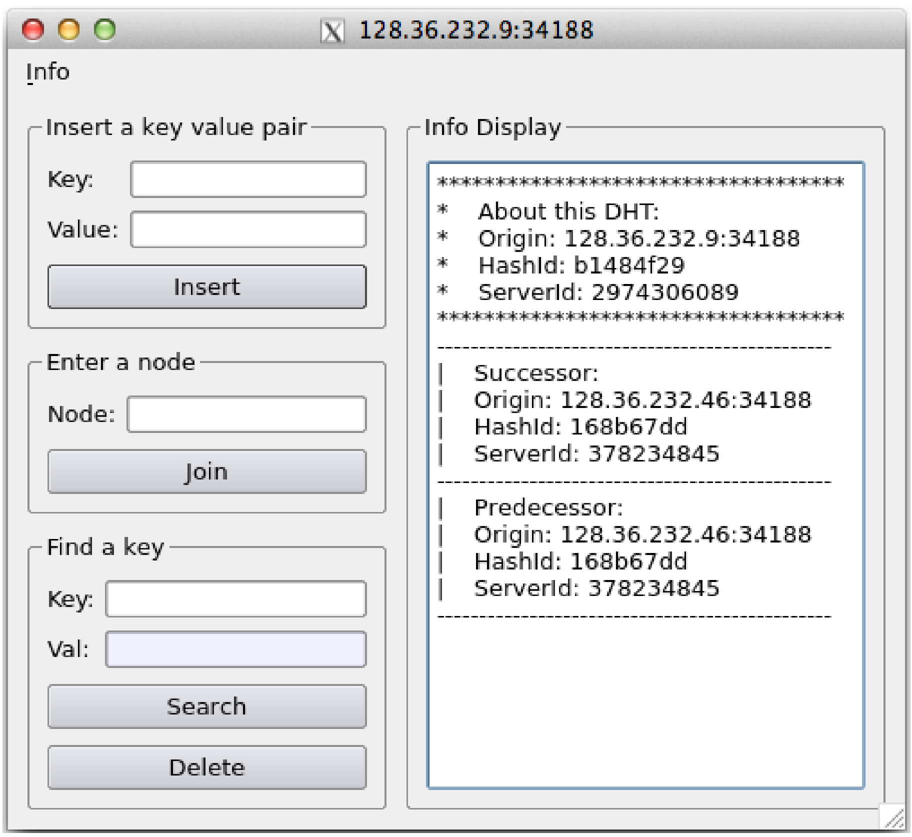
   (c) Load balancing (Key migration when a new DHT server joins and leaves the chord).

   (d) Finger table maintainance.

   (e) Key value pair search and delete.

   (f) Key value buffer on local DHT server.

2. **Overview:**



   Each DHT server provides people an interface to interact with other DHT servers and do key oprations; for example, inserting a key value pair, searching a key, and deleting a key.

3. **Design issues:**

   (a) How to map each DHT server onto the chord.
   Solution:

i. MD5 to hash the origin and the key to get a 128 bits integer.

ii. 32 bits are selected and appended from predefined positions of the MD5 result to get HashId.

(b) What to track in each DHT:
Solution:

i. Its successor and predecessor.

ii. Its finger table.

iii. Key value pairs that should be stored. Tipically, keys that are larger than the local DHT server's predecessor's server id and smaller than the local server id will be stored.

(c) What happens when a DHT A wants to join the chord:
Solution: It sends a *joinMessage* to a known DHT server B with the following format:

```
joinMessage = {
  "JoinRequest": true,
  "Origin": "128.36.232.46:34188", # A's Origin
  "HashId": "168b67dd", # A's hashId
  "ServerId": "378234845" # A's serverId
}
```

When B receives the join request message, it will first check if A has come to the right place. If not, it will forward the join request message; otherwise, B will send a *joinAcceptedMessage* to A with the following format:

```
joinAcceptedMessage = {
  "JoinRequestAccepted": true,
  "Pred": {
    "Origin": B->Origin,
    "HashId": B->HashId,
    "ServerId": B->ServerId,
  },
  "Succ": {
    "Origin": B->Succsessor->Origin,
    "HashId": B->Succsessor->HashId,
    "ServerId": B->Succsessor->ServerId,
  }
}
```

When A is notified that its join request has been accepted, it will first update its new predecessor and successor, and then sends a *UpdateSuccRequest* to its new predecessor, and a new *UpdatePredRequest* to its new successor with the following format:

```
updateSuccMessage = {
  "updateSuccRequest": true,
  "Origin": A->Origin,
  "HashId": A->HashId,
```

```
  "ServerId": A->ServerId
}

updatePredMessage = {
  "updatePredRequest": true,
  "Origin": A->Origin,
  "HashId": A->HashId,
  "ServerId": A->ServerId
}
```

A's new neighbours will update their successor and predecessor respectively after receiving the update neighbour requests.

(d) How are the keys splitted when a new DHT server A joins.
Solutions: Each key is hashed using the same way as a DHT server.

Assume that B is A's new predesessor and C is A's new successor, now some of the keys that are stored on C might be splitted to A.
A will send a *migrateKeysMess* to C with the following format:

```
migrateKeysMess = {
  "MigrateKeys": true,
  "Origin": A->Origin,
  "HashId": A->HashId,
  "ServerId": A->ServerId
}
```

Now after C receives the message, it will find all the keys that are smaller than A's server id, and send them to A.

(e) How are the keys spreaded out when a new DHT server A leaves.
Solutions: This is easier, A will simply send all the keys to its successor. In order to do this, A will send a *KVInsertRequest* for each key with the following format:

```
KVInsertRequest = {
  "KVInsertRequest": true,
  "Key": "apple",
  "KeyHashId": "5c278f10",
  "KeyId": "1546096400",
  "Val": "five"
}
```

After receiving the request, A's successor should have all the keys that were stored on A.

4. Design of Routing Protocol

1. Initialize a finger table

   When a node is born, it will have a default finger table. Since the chord space is 2^32, this table contains 32 entries. Take itself as the next hop node. Each nextHopNode contains 5 messages:

   initFingerTable() {

   Index = i;

   nextHopNode.Origin = localOrigin;

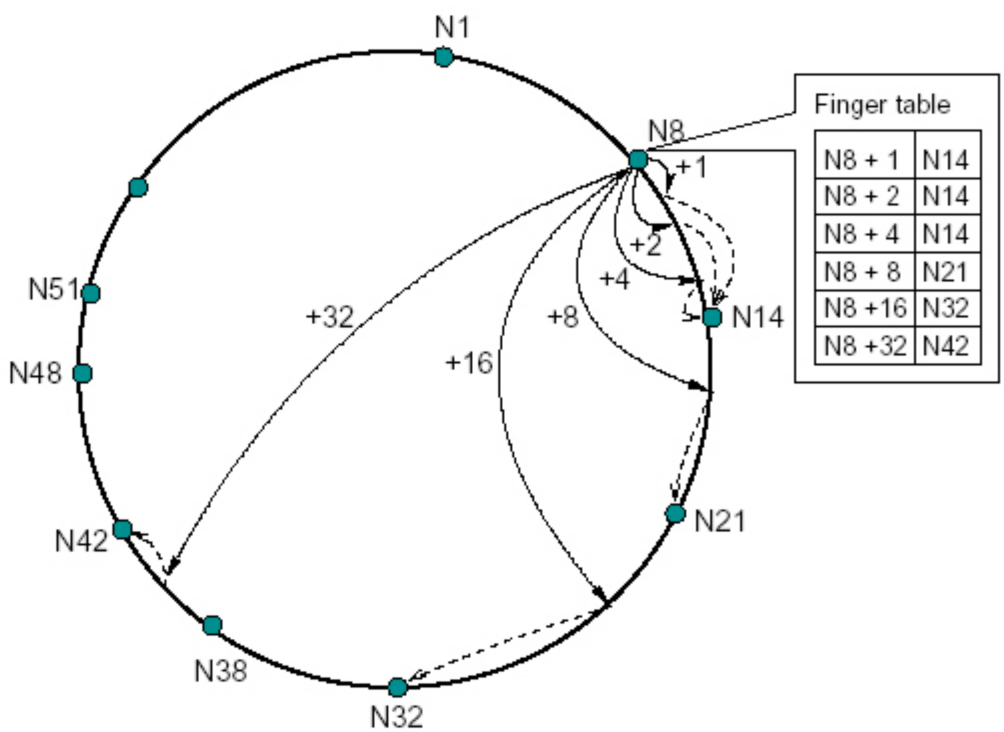
   nextHopNode.HashId = hashId;

   nextHopNode.ServerId = serverId;

   nextHopNode.HopPoint = serverId + 2^{i-1} ;

   nextHopNode.Distance = clockwise distance from current hopPoint to the node's serverId;

   }



2. Update the finger table when node joins and leaves:
   Node Join:
   - Node M wants to join the Chord by sending DHT Server a join request. After this request is accepted and M is added into the chord, M sends its successor N an ***UpdateFingerTableMessage***.

     UpdateFingerTableMessage {
     "Direction": "join";
     "Origin": M's localOrigin;
     "HashId" : M's hashId;
     "ServerId" : M's serverId;
     }

   - N receives ***UpdateFingerTableMessage*** and search through its local finger table to see which entry should be updated. N calls function ***updateFingerTable*** and send back to M ***UpdateNewFingerTableMessage*** which contains N's info.

     Then N forwards the ***UpdateFingerTableMessage*** to its successor. Forwarding halts when the successor happens to be M.

     updateFingerTable {
     for i in 32:
     M.distance = clockwise distance from current hopPoint to the M's serverId
     if N.fingertable[i].distance > M.distance
     update the node info of ith entry to M;
     }

     UpdateNewFingerTableMessage {
     "Direction": "join";
     "Origin": N's localOrigin;
     "HashId" : N's hashId;
     "ServerId" : N's serverId;

     }

- M receives ***UpdateNewFingerTableMessage*** and calls ***updateFingerTable*** function as N does. No forwarding this time.
- By this procedure, we update the finger table of all nodes in Chord.

Node Leave:

- DHT Server M wants to leave the Chord. M sends to its successor N ***UpdateFingerTableMessage.***

```
UpdateFingerTableMessage {
        “Direction”: "exit";
        “Origin”: M’s localOrigin;
        “HashId” : M’s hashId;
        “ServerId” : M’s serverId;
        “Successor” : M.successor;
}
```

- N receives ***UpdateFingerTableMessage*** and calls ***updateFingerTable.*** Then N forwards this message to its successor. Forwarding halts when the successor is empty or happens to be M.successor (in this scenario, it is N itself)

```
updateFingerTable {
  for i in 32:
     if N.fingertable[i].Node == M
        N.fingertable[i].Node = M.successor;
}
```

- By doing so, we update the finger table of all nodes left in the Chord.

3. Operating on the keys

Key search:

- Search a key on DHT Server N:
  N first searches locally. If the key is not local, we find out the next node to hop calling function ***searchFingerTable***. Then N sends the node ***KeySearchRequestMessage.*** Search halts when ***KeySearchRequestMessage.HopLimit*** == 0.

```
searchFingerTable(keyId) {
  for i in 32: {
        if (keyId > N's serverId + 2^(i-1)) && (keyId < N's serverId + 2^i)
                return N.fingertable[i]. nextHopNode;
  }
  return N.fingertable[32]. nextHopNode;
}

KeySearchRequestMessage {
     "KVSearchRequest"= true;
      "KeyId" = keyId;
      "Origin" = N's Origin;
      "HopLimit" = 32;  // HopLimit-- for each forwarding, in case the
                                   key is not in the storage
}
```

Key Delete:

- Works similarly as search.

4. Build a Cache for recently searched key, so next search of these "Hot" keys will be faster.

   Build the cache:

   - Each time when DHT Server N search a key, an entry containing the keyId and the server address where the key is located on is added to the Cache.
   - When performing the search next time, we first search locally and then search the cache.

   Maintain consistency:

   - What if the key the cache contains is deleted?
   - When search through the cache fails, we will delete this entry in local buffer.