P2P with Chord

Distributed System Project | PoliMi

29th March 2019

Giuseppe Severino, Marco Mussi

Design Choices

- Number of bits in the key/node identifier **m = 32**, chosen in order to make the probability of two nodes or keys hashing to the same identifier negligible (2^{32} different IDs)
- **Consistent hashing** is implemented by assigning each node and key an m-bit identifier using **SHA-1**. It is obtained by hashing the node's IP and Port address for the node, while a key identifier is produced by hashing the key (as specified by the original paper)
- Adoption of the **socket** protocol to handle the communications between nodes

Client Class

It asks to the user to **select** the operation to perform:

- **Create**
  - Input: local destination port
  - Output: the node is added to a new Chord ring

```
// create a new Chord ring.
n.create()
    predecessor = nil;
    successor = n;
```

- **Join**
  - Input: local destination port + ip and port of the chord ring to connect to
  - Output: the node is added to the specified Chord ring

```
// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);
```

- **Search**
  - Input: ip and port of a chord node + the key to search
  - Output: the node in which the key is contained

Node Class

It contains all the characteristics of a **Chord node**

The **Finger Table** of a node is stored in an **HashMap**

Each node keeps track of its successors with an ArrayList of size 32, in order to handle the **failures** in the ring

It implements the **findSuccessor** and the **closestPrecedingNode** methods

The following **threads** are **started** after the node initialization:

- Listener
- Stabilize
- FixFinger
- CheckSuccessor

Listener Thread

- Is an always active **ServerSocket** that accept request from the network
- It creates a new ManageRequest thread for every incoming request
- The **ManageRequest** thread analyze the received object, understand the request type and sends the response to the sender coherently

Stabilize Thread

- This thread **checks** periodically if the **predecessor** has **failed**, verifies the node immediate **successor** and tells the successor about the existence of the current node. At the end, a notify message is sent to successor.

```
// called periodically. checks whether predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```

```
// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';
```

```
// called periodically. verifi es n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);
```

Fix Finger Thread

- This thread **checks periodically** the entries of the **finger table**, to make sure they are correct

```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find successor(n + 2^(next-1));
```

Check Successor Thread

- This thread **checks** periodically if the **successor is still alive**. If a node's immediate successor does not respond, the node can substitute the **second** entry in its **successor list**

Request

- The **Request** class is an **abstract class** used for all the socket **communications** through the nodes. This class is implemented by the **concrete subclasses**. They are:
  - CheckStatusRequest
  - GetPredecessorRequest
  - GetSuccListRequest
  - NotifyRequest
  - UpdateSuccessorListRequest