CSE 513 Project 1 Report

Distributed Peer-to-peer System

Abstract

As a part of the CSE 513 class, the prototype of a peer-to-peer system has been designed. In this project, each peer is capable of locating and retrieving data hosted in the system without the guidance of the centralized directory service. The peer is multi-threaded in order to achieve high performance and simplicity.

Athichart Tangpong
Hendra Saputra

Table of Content

1. Overview

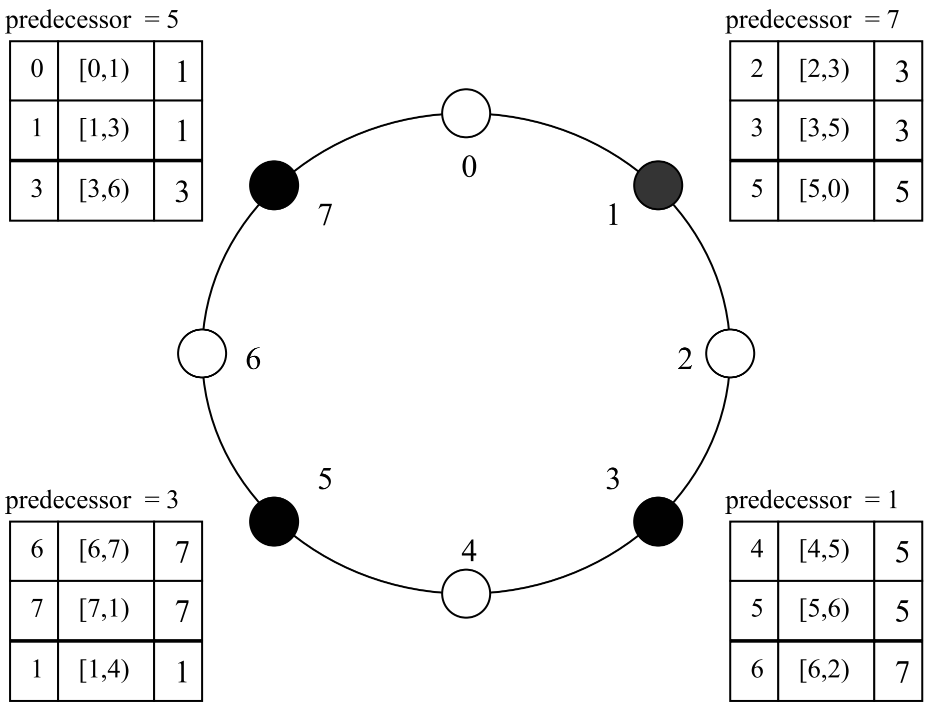

Figure 1. Overview of the system

In this project, each peer maintains a table, called finger table, initialized and updated upon the joining of a new peer, shown in Figure 1. In order to locate and retrieve the data associated with a key, the requesting peer needs to locate the peer that is hosting the keys first, by using the protocol described in 2.1.

2. Protocol

The protocol used in this project can be divided into three major parts: locating the successor of a identifier, joining, and handling peers' leave and abnormal termination.

2.1) Finding the successor of a key

In order to locate the successor of an identifier, the peer executes the pseudo code described in Figure 2. The only requirement that guarantees the correctness of the protocol is that the predecessor and the successor rings must be correctly maintained.

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

n.**find_predecessor**(id)
   n' = n;
   **while** (id $\notin$ (n', n'.successor])
      n' = n'.closet_preceding_finger(id);
   **return** n';

n.**closet_preceding_finger**(id)
   **for** I = m **downto** 1
      **if** (finger[i].node $\in$ (n, id))
         **return** finger[i].node;
```

Figure 2. The pseudo code to locate the successor of an identifier *id*

2.2) Join the system

When a peer want to join the system, it executes the pseudo code in Figure 3. This pseudo code is a bit different from the one provided by [1]. The differences are made inclined and bold.

```
#define successor finger[1].node
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    init_finger_table(n');
  else
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m -1
    if (finger[i+1].start ∈ [n, finger[i].node)
      finger[i+1].node = finger[i].node;
    else
      finger[i+1].node = n'.find_successor(finger[i+1].start);

n.update_others()
  for i = 1 to m
    if (n != find_successor($n-2^{i-1}$))
      p = find_predecessor($n-2^{i-1}$)
    p.update_finger_table(n, i);

n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor;
    p.update_finger_table(s, i)
```

Figure 3. The pseudo code for the node join operation

2.3) Handle peer leave or abnormal termination

To handle node failure or leave, the peer 1 periodically send a probe message, specifying the initiator which is 1, along the predecessor ring. If the message comes back to it the initiator, that means the ring is not broken and all peers are functional. If a peer finds that its successor fails, it sends a probe return message, specifying the initiator and the failed node, back via the successor ring. The peer that discovers that its successor is the dead node or fail changes the predecessor of the initiator of the probe return message to it and update its successor to the initiator. Moreover, upon receiving a probe return message, each peer changes its finger[i].node which is the failed node to the initiator.

For example, suppose the peer 5 fails, shown in Figure 4. The peer 1 sends a probe message along the predecessor ring. When the message reaches the peer 7, it discovers that its predecessor has failed, then send a probe return message back via the successor ring. When the probe return message arrives at the peer 3, it finds that the failed node is its successor, then update the peer 7's predecessor and its successor.

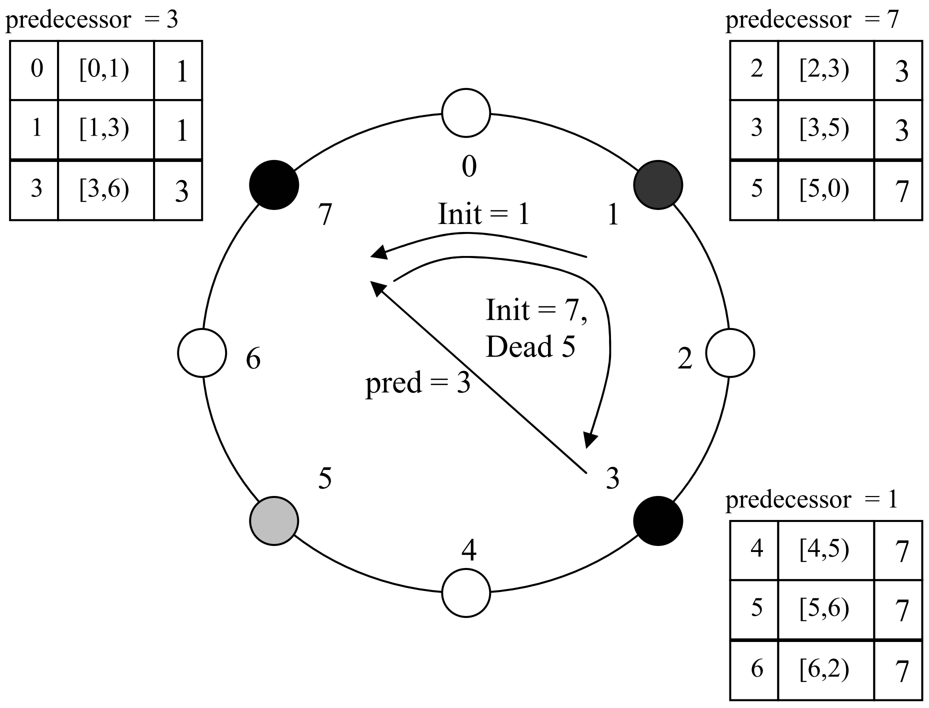

Figure 4. Update finger tables, the predecessor and the successor rings

3. Design

Each peer is equipped with the simplicity and performance of multi-threading technique. Upon the arrival of a new request, the peer creates a new thread to service it. The thread's operation depends on the type of the request, extracted from the OPCODE field in the message.

Basically, each peer has two running threads, one for handling in-coming requests and the other for processing users' request. To handle the case that some nodes fail or leave the system, peer 1 has an additional thread running in background to re-establish the predecessor and successor rings, as well as correct the finger tables of all nodes.

As far as the keys are concerned, peer 1 initially hosts all the keys and is the first peer on-line. When a new node joins the system, the keys, along with their associated data, are transferred to their appropriate successors. In contrast, when a node is going to leave the system all its keys are migrated to an appropriate successor.

4. Example

<u>After the peer 3, 5, and 7 joined :</u>

```
kronos> pka
Node = 1 (kronos)
Pred = 7 (scalosian)
Start   Begin   End     ID      Host
```

```
2      2      3      3      ubese
3      3      5      3      ubese
5      5      1      5      lumati

My Key List: 1 8

Finger Table of Node: 3
=====================================
Start  Int_Start  Int_End  Successor
4      4      5      lumati(5)
5      5      7      lumati(5)
7      7      3      scalosian(7)

Finger Table of Node: 5
=====================================
Start  Int_Start  Int_End  Successor
6      6      7      scalosian(7)
7      7      1      scalosian(7)
1      1      5      kronos(1)

Finger Table of Node: 7
=====================================
Start  Int_Start  Int_End  Successor
0      0      1      kronos(1)
1      1      3      kronos(1)
3      3      7      ubese(3)
```

Retrieve the data associated with the key 4

```
kronos> g 4
get data for key number: KEY: 4
Title : One night only
ARTIST : Bee Gees
COUNTRY : UK
COMPANY : Polydor
YEAR : 1998
That key (4) is located in lumati (node id: 5)
```

After the peer 5 left :

```
kronos> pka
Node = 1 (kronos)
Pred = 7 (scalosian)
Start   Begin   End     ID      Host
2       2       3       3       ubese
3       3       5       3       ubese
5       5       1       7       scalosian

My Key List: 1 8

Finger Table of Node: 3
=====================================
Start  Int_Start  Int_End  Successor
4      4      5       scalosian(7)
5      5      7       scalosian(7)
7      7      3       scalosian(7)

Finger Table of Node: 7
====================================
Start  Int_Start  Int_End  Successor
0      0      1       kronos(1)
1      1      3       kronos(1)
3      3      7       ubese(3)
```

Retrieve the data associated with the key 4 again

```
kronos> g 4
get data for key number: KEY: 4
Title : One night only
ARTIST : Bee Gees
COUNTRY : UK
COMPANY : Polydor
YEAR : 1998
That key (4) is located in scalosian (node id: 7)
```

5. Comment

The system is fully operational with a minor limitation. In this version, a new thread is created upon the arrival of a new request. Unfortunately, the PTHREAD implementation allows only 2039 thread creations. Then each peer can accept approximately 2039 connections. This nuisance can be eliminated easily by having each peer adopt the thread-pool model, rather than on-demand thread creation.

6. Reference

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaasshoek and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM'01