EE122 Project 2B: P2P *tiny* World of Warcraft

Instructor: Prof. Ion Stoica (istoica@eecs.berkeley.edu)

TAs: Junda Liu (liujd@eecs.berkeley.edu)
DK Moon (dkmoon@eecs.berkeley.edu)
David Zats(dzats@eecs.berkeley.edu)

Due: 11:50pm on December 11, 2009 through bspace

Version History:
0.3 Change error code of P2P_BKUP_RESPONSE. Bonus has 10 points.
0.2 Correct section 3.3 bullet 1. Delete section 4 bullet 5. Add new section 4 bullet 5.
0.1 First release

1. Overview

In part A, you solved the scalability problem by partitioning workload among servers. As you deploy many servers, hardware failures are inevitable. And if a server crashes, all user data on that server are lost (assuming hard drive also fails if user data are stored in files).

In this part, you will build a P2P data storage between servers to solve this problem. The basic idea is simple: duplicate and distribute. Each server makes copies of its data and sends them to other servers as backup. It also sends update messages to keep backups up-to-date.

The client is the same as part A. The tracker is almost the same except the hash function is slightly modified. To reduce workload, you are **NOT** required to modify your tracker. You can just use the reference tracker.

You are required to add following features to server:
1. Calculate P2P_ID from IP and TCP port
2. Connect to other servers' TCP port, and send/receive P2P messages
3. Receive P2P messages from TCP port and process them
4. Detect backup server failure and find another server as backup

2. Details

Servers use a simplified P2P network to exchange messages. When a server starts, in addition to procedures described in project 1 and project 2 part A, it computes a unique P2P_ID from its IP address and TCP port, and joins the P2P network. Every server stores some user data and a copy on its backup server, so it can recover user data after crash.

2.1 P2P_ID

Every server has a unique P2P_ID, and servers talk with each other using this P2P_ID. The P2P_ID space is [0, 1023]. If a server realizes it has the same P2P_ID with an existing server, it must exits (More details in Section 2.4). For every player, there is also

a P2P_ID calculated from player name using the same function. Each server decides which player data it should store based on its own P2P_ID and player's P2P_ID.

Calculate P2P_ID:

<table>
<tr><td>Input</td><td>Player: unsigned char pointer of player name string.
Server: 7-byte unsigned char array, first 4 bytes are non-local IP address, following 2 bytes are TCP port number, last byte is always 0x00.
Note: 127.0.0.1 is local IP and should NOT be used.
Example: IP=128.32.42.197, TCP port=12345, the array in hex format:

<table><tr><td>0x80</td><td>0x20</td><td>0x2A</td><td>0xC5</td><td>0x30</td><td>0x39</td><td>0x00</td></tr></table></td></tr>
<tr><td>Function</td><td>Slightly modified hash function from part A:
unsigned int calc_p2p_id(unsigned char *s) {
unsigned int hashval;
for(hashval=0;*s!=0;s++)
hashval=*s + 31*hashval;
return hashval % 1024;
}
The difference is shown in bold.
Note:When s is a pointer to player name string, this function calculates player's P2P_ID.</td></tr>
<tr><td>Output</td><td>P2P_ID of server or player, an unsigned integer id and 0≤id≤1023</td></tr>
</table>

2.2 Primary and backup server

Every player has a primary and a backup server to store its data.

Select primary server:

1. Use calc_p2p_id(player name pointer) to get the player's P2P_ID, assume it's *idx*
2. Among the servers whose P2P_ID is not smaller than *idx*, the one with smallest P2P_ID is primary server
3. If *idx* is bigger than all server P2P_IDs, the server with smallest P2P_ID is the primary server

Below is an example. Numbers inside the circle is the P2P_ID of that server. The circle is only for illustration, and doesn't imply any actual toplogy or transmission.
If *idx* is 56, then server with P2P_ID 123 is the primary server.
If *idx* is 456, then server with P2P_ID 456 is the primary server.
If *idx* is 1010, then server with P2P_ID 123 is the primary server.

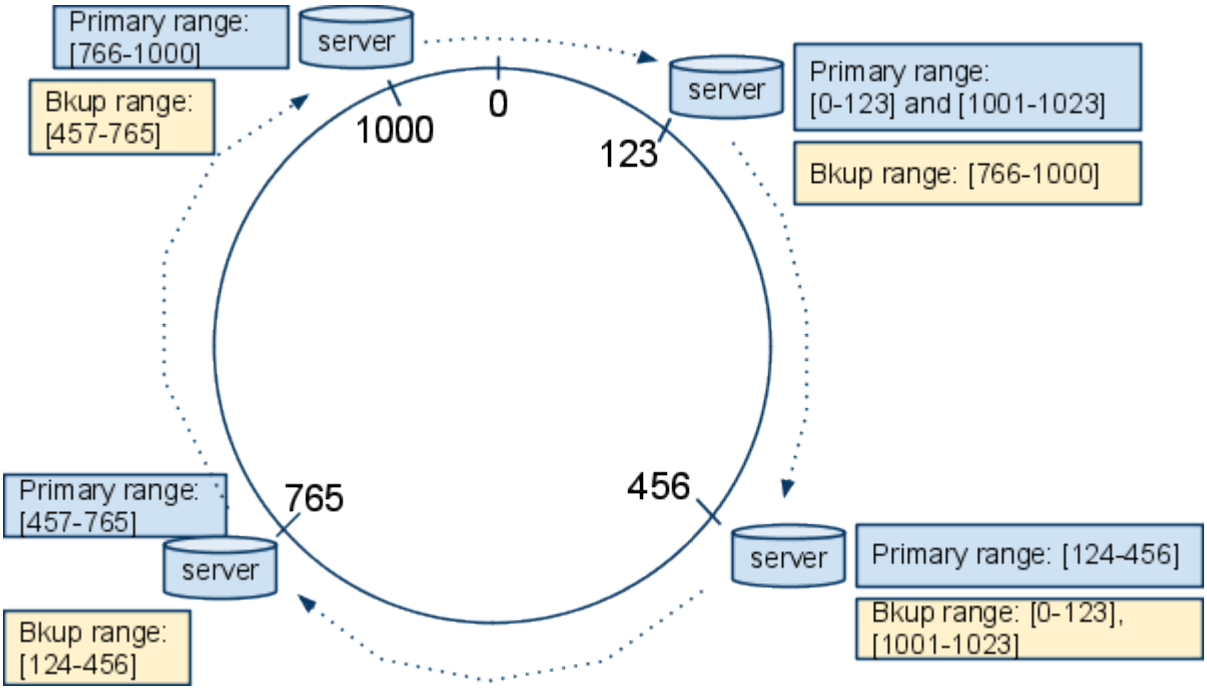
Backup server:
Every server finds a backup server for its data. For server with P2P_ID *x*:

1. If no other server exists, do not use backup server
2. Among all servers that have P2P_ID bigger than *x*, the one with smallest P2P_ID is backup server
3. If all other servers have smaller P2P_ID, the one with smallest P2P_ID is backup server

As shown below, server 456 is the backup server for server 123, and server 123 is backup server for server 1000. If server A is B's backup server, we call A is B's **successor**, and B is A's **predecessor**. When P2P network only has 2 servers, they are both successor and predecessor to each other. TCP connection for backup is always from predecessor to successor.

Following above principles, each server has a primary range and a backup range. If A is B's successor, then A's primary range is:

- $[B_{id}+1, A_{id}]$, if $B_{id} < A_{id}$
- $[B_{id}+1, 1023]$ and $[0, A_{id}]$, if $B_{id} > A_{id}$



2.3 Server goes up/down

When server A comes up (either first time or after a crash)
A joins P2P network by following steps:

- A reads *peers.lst* and finds predecessor B and successor C
- A connects to B and C and sends P2P_JOIN_REQUEST (If B and C are the same server, A only establishes one connection and send one request)
- A receives P2P_JOIN_RESPONSE from B and C
- A closes connection to B but keep the connection to C

B does the following:

- B sends back P2P_JOIN_RESPONSE containing all user data in its primary range
- B closes connection to previouse successor
- B connects to A's TCP port and keeps the connection

C does the following:

- C modifies its primary and backup range
- C finds which players now should be primarily stored on A
- C sends back P2P_JOIN_RESPONSE containing players from previous step

If B and C are the same server, the server does C's steps first, then B's steps. So it sends two P2P_JOIN_RESPONSE messages back to A.

When server A fails:

- Its successor C will take requests (assume tracker can detect this and return the backup server to client. See Section 5 for more details). But for simplicity, backup server doesn't change its primary or backup range.

- Its predecessor B detects this because of broken TCP connection. It reads *peers.lst* and follows above principles to find another server (usually C) as backup. Then it connects to the new backup server and sends P2P_BKUP_REQUEST when necessary.

The primay and backup range on every server will change **only** when other server joins P2P network. They stay the same when server fails. In this design, data will be still lost if two P2P_ID-adjacent servers both fail.

2.4 What's new when server starts

In addition to listening on TCP and UDP ports, the server gets its non-local IP address, computes the P2P_ID, and tries to read a file *peers.lst*. *peers.lst* may be a local file or stored on NFS and shared among servers. It contains one server information per line, see below for example.

If *peers.lst* doesn't exist, it means the server is the first in the P2P network. It creates *peers.lst* file and writes its "P2P_ID IP TCPport\n" (space delimited, without quote) into the file. IP address uses dotted-quad notation. Example: 436 192.168.1.100 12345

If the file exists, the server reads the file and checks all P2P_IDs in it.

- If the server doesn't find its P2P_ID in the file, it appends its "P2P_ID IP TCPport\n" to *peers.lst*, and sends messages to join P2P network.
- If it finds same P2P_ID, but IP or TCPport is different, server prints out "P2P_ID conflicts." and exits.
- If it finds itself in the file, it sends messages to join P2P network.

Packet formats are described in Section 3.

3. Protocol

All P2P messages are TCP. MsgVer is one byte, and always 0x04.

3.1 P2P_JOIN_REQUEST

MsgVer(1B)=0x04	MsgLength(2B)	MsgType(1B)	Server P2P_ID(4B)

1. Sent from new coming server to its successor and predecessor.
2. MsgLength: 0x08
3. MsgType: 0x10
4. Server P2P_ID: network order P2P_ID of the sender

3.2 P2P_JOIN_RESPONSE

MsgVer(1B)=0x04	MsgLength(2B)	MsgType(1B)	User Number(4B)	User Data(20B)	....

1. Sent from existing servers to new coming server
2. MsgType: 0x11
3. User Number: how many users this message contains, network order unsigned int.

4. User Data:20 bytes user info, same as project 2 part A:

Name(10B)	HP(4B)	EXP(4B)	X(1B)	Y(1B)

5. One message may contain multiple user data, and spans multiple TCP packets
6. When server receives this message, it updates local user data

3.3 P2P_BKUP_REQUEST

MsgVer(1B)=0x04	MsgLength(2B)	MsgType(1B)	User Data(20B)

1. Sent from a server to its successor to store or update user data. A server needs to send this message when it gets SAVE_STATE_REQUEST (defined in part A) from client.
2. MsgLength: 0x18
3. MsgType: 0x12
4. User Data: 20 bytes, same as above

3.4 P2P_BKUP_RESPONSE

MsgVer(1B)=0x04	MsgLength(2B)	MsgType(1B)	ErrorCode(1B)	Padding(3B)

1. Sent from successor to predecessor, in response to P2P_BKUP_REQUEST.
2. MsgLength: 0x08
3. MsgType: 0x13
4. ErrorCode:
    1. Success: 0x00
    2. Error: 0x02

4. Evaluation

1. Tracker will NOT be tested in this part, unless you implement bonus features.
2. Game logic will NOT be tested in this part.
3. Test script will frequently start and kill multiple server processes.
4. Test script sends SIGTERM signal to kill server process. It's best that your server can handle this signal and exit gracefully.
5. Same as part A, your sever should read/write "*users/playername*" file for player data

5. Bonus (10 points)

To reduce workload, we assume that tracker can detect server status and return the backup one to client if primary server fails, but current tracker doesn't support this. Your team can implement the following for extra credit (10 points):

1. Tracker reads *peers.lst* and uses calc_p2p_id(player name) to decide which server to return. (Reference tracker already implements this)
2. Heartbeat message from tracker to server or other method to detect failed server
3. If primary server fails, tracker will return backup server to client

If your team implement bonus features, please explain how they are implemented in your submitted readme file.

6. Appendix

6.1 Get non-local IP address

Call gethostname(name), then gethostbyname(name).
h_addr_list[0] has IP address and inet_ntoa converts it to dotted-quad string.