CS 425/ECE 428
Swapnil Ghike(ghike2) / Wonho Choi(wchoi20)

Design Document for MP2

The goal of this machine problem is to implement a chord system where all the nodes are running on the same system. The chord system implemented is capable of adding nodes, adding files, deleting files, looking up files and getting the finger and key tables for each node. The chord system supports concurrency operations where operations are executed on different nodes at the same time. Key assumptions are we are not deleting nodes from the chord system and files that are to be stored are small enough to be saved on the memory.

The whole implementation will be divided into two programs. One is the listener and the other is the node. The listener will create a new process whenever it receives the ADD_NODE command from the user. This will be only allowed when the listener is connected to the introducer node. The introducer is a node that acts like the leader of the chord system. The chord system always starts by the initiation of the introducer node(id = 0). The introducer is the only node that is capable of adding additional nodes to the system. The listener acts as a gateway that passes on user inputs to one of the node that is inside the chord system and displays results of the related commands.

1. What we learnt while using Thrift:
Thirft is an interesting and an easy to use framework for RPC invocation. The learning curve required to implement this MP was quite small. We needed to be careful about the following aspects though –

- That a server can also act as a client did not strike us immediately. We needed some thinking to decide to implement a client and a server in the same file.
- It is critical to keep an eye on which data belongs to other node. To access this data, one needs to make a RPC call.
- Avoiding RPC calls on the local server is a good idea.
- We provided concurrency and avoided deadlocks by using TThreadedServer. We needed to check up the definitions of all of TSimpleServer, TThreadServer and TNonblockingServer. We decided to avoid TNonblockingServer because it has a fixed number of threads , and in a large system, all of them can deadlock at the same time in some cases. TThreadedServer was the obvious choice to avoid deadlocks caused due to cyclic RPC call graph, because it creates a new thread for every client connection. If the number of connections becomes too large, then one can look at solutions like TThreadPoolServer, which creates a new thread for every connection and limits the number to a maximum. This problem also does not have the IO bottleneck caused by the single thread dedicated to network IO in TNonblockingServer, however deadlocks need to be avoided very carefully here with some additional scheme.

2. How the listener/node picks ports to use:
All nodes are run as separate processes on the same machine. Therefore, only the port that the node is listening on will be used to distinguish the location of each node. The listener will be responsible of the calculation of the ports that the nodes will be using. The ports will be calculated randomly unless the user specifies a port number that a node should try first by the --startingPort option in the listener. When calculating random ports, the calculated port range will be between 49152 and 65535. Registered ports and well-known ports will not be included in the random ports. After a random port has been selected, the port will be checked if it is available by using socket programming. A socket will be created and bind() function will be called. If bind() returns -1, then the port is not available. In other cases, the port is available and we free the port from bind and pass it onto the node program's -port option.

3. Special cases added to the algorithms in the chord paper:

We set the predecessor and successor of a node to the introducer node as soon as the new node joined the introducer node, this allowed us to simplify our implementation. Similarly, when the introducer node boots up, it sets itself as its own predecessor and successor. Fix fingers does not update 1st finger entry, so we update it every time a successor pointer changes. The pseudo code is as follows (our modifications in red) –

```
// ask node n to find id's successor
n.find_successor(id)
        n' = find_predecessor(id);
        if (Id = n)
                ret = n.successor;
        if (n = 0 and n.successor = 0)
                n.successor = id;
                n.predecessor = id;
                n.finger_table[1] = n.successor;
        if (Id = n)
                ret = n.successor;
        return n'.successor;
```

find_predecessor can enter an infinite loop if the node id tries to find the predecessor of id. This is fixed as follows –

```
//ask node n to find id's predecessor
n.find_predecessor(id)
        If (Id = n)
                return n.predecessor;
        n' = n;
        while (id ∉ (n', n'.successor])
                n' = n'.closest_preceding_finger(id);
        return n';
```

4. How nodes transfer keys:

We also check if there are any files to be moved to predecessor every time the predecessor notifies of its own presence to node n. Note that this check is necessary every time since there could arise a situation when the a thread in node n is moving files to predecessor and another thread in n stores a file in n that was actually meant for n.predecessor. This can typically happen when the system has not stabilized. This is implemented as follows –

```
// n' thinks that it might be our predecessor.
n.notify(n')
        if (predecessor = nil or n' ∈ (predecessor, n))
                predecessor = n';
        if (there are any keys to be moved to predecessor)
                predecessor.move(keys_table);
```

5. How we provide concurrency:

- We provided concurrency and avoided deadlocks by using TThreadedServer. TSimpleServer runs a single server thread and it can deadlock if it makes a client call on another server and the two servers form a cyclic RPC call graph. We decided to avoid TNonblockingServer because it has a fixed number of server threads, and in a large system, all of them can deadlock at the same time in some cases. TThreadedServer was the obvious choice to

implement concurrency and avoid deadlocks caused due to cyclic RPC call graph, because it creates a new thread for every client connection. If the number of connections becomes too large, then one can look at solutions like TThreadPoolServer, which creates a new thread for every connection and limits the number to a maximum. This problem also does not have the IO bottleneck caused by the single thread dedicated to network IO in TNonblockingServer, however deadlocks need to be avoided very carefully here with some additional scheme.

- There are two other threads running in a node to stabilize pointers and fix finger table periodically. These threads can modify finger table and keys table in the node at the same time when the server thread is reading the finger table or keys table. Thus we use a mutex lock to protect accesses to successor, predecessor and finger table pointers and another lock to protect accesses to keys table. The order of acquiring locks is the same, so they don't cause any deadlock.

6. Message formats and object structures:

- We define a finger entry as struct finger_entry {int id; int port;}. This is the most commonly passed structure in messages.
- We define another structure _FILE {string filename; string filecontent;}. This structure is used as a parameter in add_file and del_file RPCs.
- We define struct node_table{vector<finger_entry> finger_table, map<string, _FILE> keys_table}. This is returned as a response to get_table RPC, it is also passed as a parameter to accept_files RPC which is indirectly called while moving keys/files to a predecessor.
- We define struct file_data{string filename, string filecontent}. This is returned as response to get_file RPC.
- Our Node object stores id, port, successor and predecessor which hare finger entries, finger_table which is a vector of finger entries, keys table which is a map of key id and _FILE structures, introducerPort, stabilizeInterval, fixInterval, seed.
- Other details are in mp2.thrift file.

7. How we tested the code:

We first tested our code by running only the node program several times on different terminals. We checked that thrift is working and every node knows the introducer port and communicates with each other. After we confirmed that thrift is working, we tested if the finger table of each node was filled in correctly. In order to test the correctness of the finger tables, the successor and predecessor of each node needs to be retrieved correctly. There were some corner cases that needed to be considered carefully. In the end, every node got its successor and predecessor in the correct manner. We tested many cases like the following and obtained the correct output after the system was stabilized –

1. When finger table has stabilized, add nodes one-by-one/concurrently.
2. When finger table has not stabilized, add nodes one-by-one/concurrently.
3. When finger table has stabilized, add files one-by-one/concurrently.
4. When finger table has not stabilized, add files one-by-one/concurrently.
5. After finger table has stabilized, add files. After keys have stabilized, add nodes one-by-one/concurrently.
6. After finger table stabilized, add files and nodes concurrently.

We have not implemented any of the Log4cxx functionality since we used stdout for debugging and logging purposes.