ECE428/CS425/CSE424: Distributed Systems Spring 2012

Machine Problem 2

Distributed Hash Table

In this MP, you will be implementing a variant of the Chord system [1]. Your main task will be to implement (i) node joins, (ii) adding a file, (iii) deleting a file, and (iv) searching for a file.

Assumptions

1. Nodes do not leave the system (no node failures)
2. The keys and the node IDs are m bits in size ($5 <= m <= 10$)
3. The files are small (tens of bytes)

High Level Requirements

1) Each node in the system must be simulated by a separate process
2) Must use sockets for all inter-process communication (e.g., via the Thrift framework)
3) Must work with a minimum of 10 nodes

The Chord system that you will design should be capable of supporting the following operations: node joins, adding a file, deleting a file, and looking up for a file. Furthermore, all these operations must support **concurrency** (e.g., 4 nodes join, 2 nodes add some file, 6 nodes delete some file, and they all might initiate their operations at the same time).

For concurrent node joins, you should implement the stabilization protocol (section 5.1 and figures 4 and 7 of the Chord paper [1] – you do not need to implement algorithms in figure 6, which are supplanted by those in figure 7). As nodes join, the nodes must transfer files as appropriate (see "Transferring keys" of section 4.4). In the steady state, each file in the system should be stored at only **one** node.

Your system will **not** be storing files onto disk: you may assume the files are small (~ tens of bytes), so you should just store them in memory.

Each node runs as a separate process. Each node has an ID, given from the command line, as well as a port, also given from the command line, to listen on. **NOTE:** all the nodes in the system run on the same host, so the "location" information (e.g., for finger pointers etc) about each node is simply the port it is listening on, instead of the IP address:port combination.

Components

The main components of the system are the Chord nodes. When a new node starts, to join the system, it must know of some existing node already in the system. In this MP, we assume that this node is always the node with ID 0. Let's call it the **introducer**. It is the first node to be started, and all new nodes will contact it to join. If a node is given the ID 0, it can assume it is the introducer and is the first node to be started. The **introducer**, besides providing the "bootstrap" for other nodes, is like any other node in the system, e.g., it must follow the Chord protocol when looking up a key.

Besides processes that represent nodes in the Chord system, you will implement another program, let's call it the **listener**, that provides the user interface: takes user commands (described below) and displays results to the user. This process should be capable of passing Chord-related commands issued by the user to the **introducer** for further action. When the **listener** starts, it should start node 0 (the **introducer**), unless it is given an existing node to "attach to."

The commands that are to be supported by your system are detailed in the following table:

Command	**Usage**	**Expected output (on screen). See the Output section for specifics.**	**Function**
ADD_NODE	ADD_NODE <node ID(s)> E.g., ADD_NODE 23 13 // Add two nodes with IDs 23 and 13.	None	This command is directly handled by the **listener**: it launches the new nodes (as new processes), giving each a unique port (see Implementation section) to listen on as well as the port the **introducer** is listening on.
ADD_FILE	ADD_FILE <filename> <data> E.g., ADD_FILE f1 this is content of file 1 // Add file with name f1 and data "this is content of f1"	See Output section	This command is handled by the **introducer** (the **listener** only passes the command to and receives the result from the **introducer**)**:** the **introducer** uses the <filename> to i) find the corresponding key, ii) follow the Chord protocol to locate the node that should store the file & transfer the file to that node.
DEL_FILE	DEL_FILE <filename> E.g., DEL_FILE f1 //Delete file f1.	See Output section	This command is handled by the **introducer**: i) find the corresponding key, ii) follow the Chord protocol to locate the node that should store the file & tell that node to delete the file.
GET_FILE	GET_FILE <filename> E.g., GET_FILE f1 //Get the information of file f1.	See Sutput section	This command is handled by the **introducer**: i) find the corresponding key, ii) follow the Chord protocol to locate the node that should store the file & ask that node for the file data.
GET_TABLE	GET_TABLE <node ID> E.g., GET_TABLE 23 //Prints the finger table of node 23 along with the keys stored at 23	See Output section	This command is handled by the **introducer**: i) find the corresponding key, ii) follow the Chord protocol to locate the node & ask the node for its finger table & keys table

Implementation

You should use a Makefile that will generate the following two binaries: `node` (for the Chord nodes---including the **introducer**) and `listener`. The node must be implemented in C++, while the listener might be implemented in a language of your choice (e.g., python), as long as it runs on the EWS machines.

Interprocess communication

Interprocess communication (node-to-node as well as listener-to-introducer-node) must be over sockets. We recommend using TCP. You have the option of using sockets directly (see http://beej.us/guide/bgnet/ for a tutorial), but we strongly recommend using the Thrift framework http://thrift.apache.org/

The Thrift framework faciliates the implementation of network services and remote-procedure calls. You describe your network service: message/object structures, remote-procedures, etc., and Thrift can generate skeleton code (in C++, python, etc.) that you can fill in to implement your network service. More **importantly**, it abstracts away the network socket communication and RPC plumbing, allowing you to focus on the Chord protocol. A Thrift tutorial/example is in `/class/ece428/mp2/thrift-example`, and more links are on the course web page.

The listener

The **listener** should figure out which ports to give to nodes that it launches. The actual values of the ports don't matter, as long as they are available so the nodes can listen on them. Working on shared EWS machines might require some trial-and-error before you can find an available port. A simple automated solution might be: the **listener** automatically picks a port number (random or from a sequence), launches the node, giving it the chosen port number, and the node will exit --- before attempting to join the system --- if that port is not available, then the **listener** can try another port number. You are free to use this or other schemes.

The **listener** must take the following arguments:

- `--startingPort` (optional): if given, this is the start of the sequence of ports the **listener** should try
- `--attachToNode` (optional): if given, this is the port that ***some*** node is listening on. The **listener** should "attach" itself to this node, without launching the **introducer**, assumed to be already running. This **listener** instance will then accept all commands listed above, except ADD_NODE, and pass them to the node it is attached to (*not* to the introducer). You can think of this as different user interacting with the Chord system via the node running on his machine.

The **listener** also takes the following arguments (described in the node section) so that it can pass them to nodes it launches:

- `--m` (**required**)
- `--stabilizeInterval` (optional)
- `--fixInterval` (optional)
- `--logConf` (optional)

The node program

The `node` program must take the following arguments:

- `--m` (**required**): the number of bits of the keys/node IDs
- `--id` (**required**): the node ID of this node
- `--port` (**required**): the port this node should listen on
- `--introducerPort` (required if id != 0): the port the introducer is listening on
- `--stabilizeInterval` (optional): the interval in seconds between invocation of the stabilization protocol
- `--fixInterval` (optional): the interval in seconds between invocation of the "fix fingers" protocol
- `--seed` (optional): to seed the random number generator

- `--logConf` (optional): configuation file for logging (see below)

You may use the code provided in `/class/ece428/mp2/exampleCode.cpp` to process these command line arguments.

NOTE: all nodes, including the introducer, run the same code. (A node uses the ID parameter given from the command line to determine whether it is the introducer or not.) This will allow a **listener** to attach to any node and interact with the Chord network through that node.

Logging

For this MP, you will be dealing with multiple processes interacting with one another, each with multiple threads. Therefore, you are encouraged to use a more advanced logging system than `printf()`. `Log4cxx` is one such package: it is thread-safe, and you can have different parts of the code log under different names ("components"), and can use the log configuration file (`--logConf` option above) to control which component is enabled without having to edit code/recompile. If you decide to use log4cxx, you can use the two provided files: `/class/ece428/mp2/log.hpp` and `/class/ece428/mp2/log.cpp`. You don't have to copy them into your working directory; you just need to "point" your compile commands to them. See `/class/ece428/mp2/logging-example` on how to use them.

Each node must log the key events with the following messages (either to standout out or, INFO level if using log4cxx):

- When it sets its initial successor after just joining:
  `node= <id>: initial successor= <node id>`

- When it updates its predecessor *to a **new/different** node ID value* – don't log if no change in the value:
  `node= <id>: updated predecessor= <node id>`

- When it updates its ith (1 <= i <= m) finger pointer (including successor, with i == 1) to a new/different node ID value – don't log if no change in the value:
  `node= <id>: updated finger entry: i= <...>, pointer= <node id>`

- When it adds a file to its memory/storage:
  `node= <id>: added file: k= <...>`

- When it deletes a file from its memory/storage:
  `node= <id>: deleted file: k= <...>`

- When it is asked to delete a file from its memory/storage that does not exist:
  `node= <id>: no such file k= <...> to delete`

- When it serves a file from its memory/storage:
  `node= <id>: served file: k= <...>`

- When it is asked to serve a file from its memory/storage that does not exist:
  `node= <id>: no such file k= <...> to serve`

Hash Function

You will be using a SHA-1 hash function for generating the key corresponding to a filename. The two files are `/class/ece428/mp2/sha1.h` and `/class/ece428/mp2/sha1.c`. Once again, you do not have to copy them.

See `/class/ece428/mp2/hash-example/key_gen_test.c`. In particular, refer to the statement `SHA1Input(&sha, argv[1], strlen(argv[1]));` to understand how to call the hash function in your code and the statement `key_id = sha.Message_Digest[4]% (1<<m);` to understand how a key is generated from a file name. The keys generated by this library will be a 160 bit SHA-1 key modulo 2^m. (Make sure that when you compile your code with this library, you include `<math.h>` and when you link your codes with this library, you need `-lm` to use math library. See `/class/ece428/mp2/hash-example/Makefile`.)

You must derive the key as done in the hash-example. For example, under m=10, `foo.c` and `bar.h` will produce keys 675 and 206, respectively.

Output

This section specifies the required output formatting for the various commands. You may use/look at the code in `/class/ece428/mp2/exampleCode.cpp` for reference.

- ADD_FILE: use/see `get_ADD_FILE_result_as_string()` function
- DEL_FILE: use/see `get_DEL_FILE_result_as_string()` function
- GET_FILE: use/see `get_GET_FILE_result_as_string()` function
- GET_TABLE: use/see `get_GET_TABLE_result_as_string()` function

You may modify these functions to match your data structures, arguments, etc, but the format of the output, for the difference cases (found vs. Not found etc) must be kept intact.

Scoring Breakdown/Test Suites

- **3 points**: listener correctly starts the introducer (and correctly attaches if given `--attachToNode`).
- **5 points**: if key derivations (from file names) in all test suites are correct. **5 points deducted** for each test suite that produced a wrong key, up to a maximum of 20-point deduction.
- **12 points**: Single node network (only introducer), m=10: for each of four (4) ADD_FILE, DEL_FILE, GET_FILE, and GET_TABLE test suites: 3 points for correct output (values and formatting), 1 point for correct values with wrong format, 0 for wrong value(s) (wrong keys count separately above).
- **55 points**: 10-node network, m=10
    - GET_TABLE: **15 points**. All 10 nodes join **concurrently** and allowed to stabilize. Full points if all 10 nodes have correct pointers (predecessors, successors, fingers etc); **-10** points if >= 5 incorrect pointers; **-15** points if >= 10 incorrect pointers. **-5** points if any wrong output format.
    - ADD_FILE: **10 points**. Stable network, add 20 files (that will produce unique keys). Full points if all are stored at correct nodes. **-5** points if >= 1 incorrectly stored files. **-10** points if >= 3 incorrectly stored files. **-3** points if any wrong output format.
    - GET_FILE: **10 points**. Stable network, get the 20 added files as well as non-existent files. **-5** points if >= 1 incorrectly retrieved files (existent file claimed as non-existent and vice versa, wrong file data, wrong location information compared to when added); **-10** points if >= 3 such inaccuracies. **-3** points if any wrong output format.
    - DEL_FILE: **10 points**. Stable network, delete the 20 added files as well as non-existent files: **-5** points if >= 1 incorrectly deleted files (existent file claimed as non-existent and vice versa,

deleted the wrong file, etc); **-10** points if >= 3 such inaccuracies. **-3** points if any wrong output format.
    - Key transfer: **10 points.** 5-node network (m=10), **concurrently** add 20 files **and** 5 nodes, allow network to stabilize: files should be transferred to correct locations. **-5** points if >= 3 files at wrong locations. **-10** points for >= 6 such inaccuracies.
- **5 points:** key events are logged as described in the Logging section
- **10 points**: Report:
    - If using Thrift, describe what you did to learn it, which aspects of Thrift you were uncertain about, and what you did to confirm one way or the other, your overall experience with Thrift.
    - Describe how the listener/node picks ports to use.
    - Describe any changes/additions/special cases you need to make to the algorithms in the Chord paper. Include pseudocode.
    - Describe how the nodes transfer keys. Include pseudocode.
    - Describe how you provide concurrency and discuss related issues.
    - Describe your message formats, object structures.
    - Discuss how you tested your code.
- **10 points**: clean, well-commented code (each function – including those in the thrift file if applicable – should document its API: arguments, assumptions, output, etc.).
- **-30 points:** for any process crash/deadlock. You will be allowed to submit a small fix (change ~5 lines, only to fix the crash).

Submission

Your submission should contain the following, all in the same directory:

1) The thrift file if you are using Thrift, all source code (those auto-generated by Thrift are optional). Please do not turn in any executable or binary files (.o files). (If you are using Thrift, we will run this command `` `thrift --gen cpp -out . <your thrift file>' ``)

2) A Makefile for compiling all your code.

3) Report (txt or PDF only, please).

From inside your work directory, run:

```
make clean
/class/ece428/Handin/handin 2
```

You may hand in multiple times up until the deadline. We will grade **only** the latest submission. Only one submission per group is necessary.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, ``Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications'', ACM Sigcomm 2001.