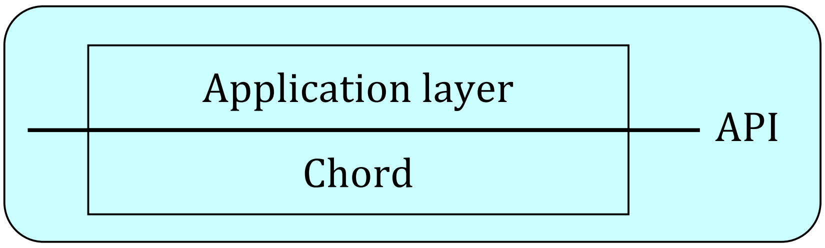
Distributed Systems Project, Spring 2013 – Assignment

In this assignment you will implement a simple distributed hash table (DHT) based on the Chord network. Your DHT should be able to maintain the Chord ring structure, store key-value pairs, and retrieve the stored values.
You will not need to implement the full Chord protocol; only the parts which are needed to ensure the correctness of the network are required. This means you only need to implement links to the immediate predecessor and successor of a node. Implementing finger tables is not a part of this assignment.
Chord provides an API for applications which wish to use its services, as shown in the Figure below.



In this assignment, your main task is to implement the Chord layer and the API it provides to the application. In addition, you need to implement a simple application which uses the API and demonstrates that your Chord is working properly.

Chord API

The API must support the following functions:

- join(host, port). This function is called when the application wants to join the Chord network. To join the network, you need to get the address of a node that is already part of the network and its port number which are passed as arguments. After the join returns, the calling node should be in its correct place on the Chord ring, with the predecessor and successor correctly informed.
- leave(). This function is called when the node leaves the network, e.g., because it is being shut down. After leave returns, the Chord ring should be intact (i.e., the old predecessor and successor are informed) and the items stored by this node need to have been passed to its successor.
- store(name, value). This function stores the value in the node which is responsible for name (more precisely, in the node which is the successor of hash(name))
- value = retrieve(name). This function returns the value which is stored under name. If there is no value stored under this name, the function returns -1. (This means you cannot store the value -1 anywhere.)

The store and retrieve operations only store simple key-value pairs. In this assignment, *names are just simple strings, the keys are the hashes of the names, and the values are integers.*

Your application should provide an interface which allows you to perform the above operations and displays the results. If you are familiar with GUI programming, you may provide a graphical interface. If you are not familiar with graphical interfaces, a simple text-based interface, where you can type in commands and see the responses printed out, is sufficient. The only requirement for full credit is some kind of a working interface; adding fancy features to the interface does not get you extra credit.

Chord protocol

The communications between the different Chord layers on different nodes is done according to the following protocol. The protocol messages are plain text and all follow the same format which is:

OP	**Arguments ...**	**<CRLF>**

Where *OP* is the operation and *Arguments* are the arguments needed for the operation (see below for details). If there are multiple arguments, they are separated by a single space. Each line is terminated with a CRLF-combination (carriage return + linefeed).
Your Chord protocol must implement the following operations:

- **JOIN**. Join takes as arguments the identifier, the IP address, and port number of the joining node. The identifier is an unsigned integer, the IP address is in dotted-decimal notation (e.g., 127.0.0.1) and the port number is an integer. The JOIN message is routed to the node which is currently responsible for the identifier of the new node.

- **JOIN_OK**. This message is sent by the node which received the JOIN and confirms the success of JOIN to the new node. As arguments, JOIN_OK has the identifier, the IP address, and port of the predecessor and the identifier, the IP address, and port of the successor for the new node (the successor is the node sending JOIN_OK and the predecessor is that node's current predecessor). JOIN_OK will be immediately followed by a TRANSFER-message which transfers the key-value pairs which are now the responsibility of the new node.

- **NEWNODE**. When a new node joins, its JOIN message is routed to the old responsible node, i.e., the new node's successor. The joining node receives the predecessor of this node in the JOIN_OK response. The joining node then sends a NEWNODE message to that node. This informs that node that it should change its successor to point to the new node. NEWNODE takes as arguments the identifier, the IP address, and port of the new node.

- **LEAVE**. This message is sent when a node leaves the network. It sends it to its own successor. The arguments for LEAVE are the identifier, the IP address, and port of the predecessor of the leaving node. A LEAVE message is immediately followed by a TRANSFER which transfers all the key-value pairs to the successor.

- **NODEGONE**. When a node leaves, it sends a LEAVE to its successor as per above. The leaving node then informs its predecessor with a NODEGONE message. NODEGONE has three arguments, the identifier, the IP address, and port of the leaving node's successor. A node receiving a NODEGONE message should adjust its successor to the node in the message.
- **STORE**. This message stores values under keys. It takes two arguments, the key (as an unsigned integer, which is the hash of the object name) and the value (as a normal integer, except -1, see above) which is to be stored under this key.
- **RETRIEVE**. This message is used to retrieve values stored in the nodes. It takes one argument, the key (as an unsigned integer, i.e., the hash of the name) of the requested object as an unsigned integer.
- **OK**. This message returns the value of the requested key as a response to RETRIEVE. It has one argument, the value as an integer.
- **NOTFOUND**. This is returned as a response to RETRIEVE if the node has no corresponding value for the requested key. This message takes no arguments.
- **TRANSFER**. This message transfers the key-value pairs from one node to another. TRANSFER has a slightly different syntax from the other nodes. The TRANSFER message takes one argument, the number of key-value pairs to be transferred. The actual key-value pairs follow after the TRANSFER command, one per line, key first and then the value, separated by a single space and terminated by a CRLF.

The above messages define the Chord protocol you implement in this assignment. The messages JOIN, STORE, and RETRIEVE must be routed through the Chord ring. The reply messages (JOIN_OK, OK, and NOTFOUND) are routed back through the Chord ring, along the same chain of nodes that the request took (see below for message routing). All other messages can be directly sent to the correct node (LEAVE, NEWNODE, NODEGONE, and TRANSFER) because its address and port are known.
Note that there is nothing in the protocol to guard against node failures. If a node crashes suddenly, it is likely that the Chord ring is broken and will not function anymore. Fault-tolerance is not required in this assignment.

Message Routing and Connection Management

Routing of the messages in the Chord ring should be done recursively. This means that when a node receives a message which is not intended for it (that is, the node is not responsible for the identifier in the message), it should forward it to its successor, which may forward it again, and so on. When the message finally has reached the intended node, it replies and the reply is forwarded back along the same chain of nodes.
Use TCP to send all messages and replies. You do not need to keep TCP connections open between a node and its predecessor or successor, since their IP addresses and port numbers are known and you can always open new connections when they are needed. For recursively processed messages (JOIN, STORE, and RETRIEVE), keep the TCP connections open for returning the reply.

Hash Functions

For getting the identifiers of nodes and objects, you can use the standard hash functions in the language you use. You are also allowed to use hash functions from external libraries. The hash values returned by the standard hash functions (e.g., in Java) are not always very uniformly distributed. If this becomes a problem, you can look for other hash functions, e.g., code provided on the Internet. Regardless of the hash functions you use, the node identifier should be the hash of the node's IP address concatenated with a colon and its port number, that is, node_id = hash("127.0.0.1:3456") if the node 127.0.0.1 is on port 3456.
Note: You *must* also implement a possibility of specifying the node and object identifiers manually at run-time. This is useful for testing that your network is working correctly, both for you and us. This could be for example implemented by a command line switch, or a button in the GUI, with the appropriate fields for entering the hashes. You are allowed to modify the API calls to achieve this.

Joining the Network

When joining the network, the new node must have the address of one node that is already part of the network. The new node then sends a JOIN message to this node, which then routes the JOIN to the correct responsible node, and the new node becomes a member of the ring.
This does not work for the very first node which joins the network. Use, for example, a command line switch to tell the starting node that it is the first node.

Storing Key-Value Pairs

You also need some kind of a simple database for storing the key-value pairs in each node.

Compatibility between Implementations

Although the protocol and API specifications should result in compatible implementations from different students, we do not require or enforce compatibility between implementations. If you have a chance to test your implementation with another student, write this in your report, along with the results of your test. We will also perform some compatibility tests during grading. If your implementation is compatible with other implementations, you (and the other student(s)) will get extra points in your grade.

Milestones

Milestone 1

Program a simple application which uses the Chord API as specified above. Application should provide a simple interface (text or GUI) and accept the command line switches defined in the assignment. The application should call the API functions, which can simply return immediately for this stage of the assignment.

Milestone 2

Implement the Chord ring construction and maintenance. This milestone requires that you implement the protocol messages JOIN, JOIN_OK, NEWNODE, LEAVE, and NODEGONE in the Chord layer. After this milestone, the API functions join() and leave() should work properly. Because there are no keys stored in the nodes, the TRANSFER message is always empty. However, you need to implement the empty TRANSFER messages for JOIN and LEAVE.

Milestone 3

Implement storing and retrieving objects. You need to implement the remaining protocol messages (STORE, RETRIEVE, OK, NOTFOUND) and complete the functionality for TRANSFER. After this milestone, you should have a working application which is able to store objects in a distributed "database" and retrieve them.

Grading

A working implementation of Milestone 2 is a minimum requirement for passing the assignment. We will grade the overall program, its correct working, as well as the style of code you write.

Hints for Testing Your Code

When testing your code and the correctness of your implementation, use at least 4 different nodes. The nodes can be on the same computer, as long as you give different port numbers to each node. Also, use the possibility to set the node and object identifiers manually, since this makes it easier to see what is going on.

Guidelines

The assignment is individual work. Every student must return their own implementation. You can of course discuss any problems you encounter with other students, but sharing code is not allowed and if found, will be considered as plagiarism.

You are free to choose any programming language, but we recommend using a higher level language, e.g., Ruby or Python, even if you have to learn the language from scratch during the assignment.

Deliverables

Program source code with documentation. The document should describe how you implemented the assignment.

Timeline

The assignment is due on February 22nd at 22:00. No extensions will be given.

Return

Return your code and documentation by email to Liang.Wang@cs.helsinki.fi as one tar-archive. Please indicate clearly your name and student ID in every source code file.