CSCI-4430/6430 Programming Languages — Fall 2019
Programming Assignment 2
Immutable Distributed Hash Table
Due Date: 7:00 PM October 28th

Version 2: Corrected SALSA arguments

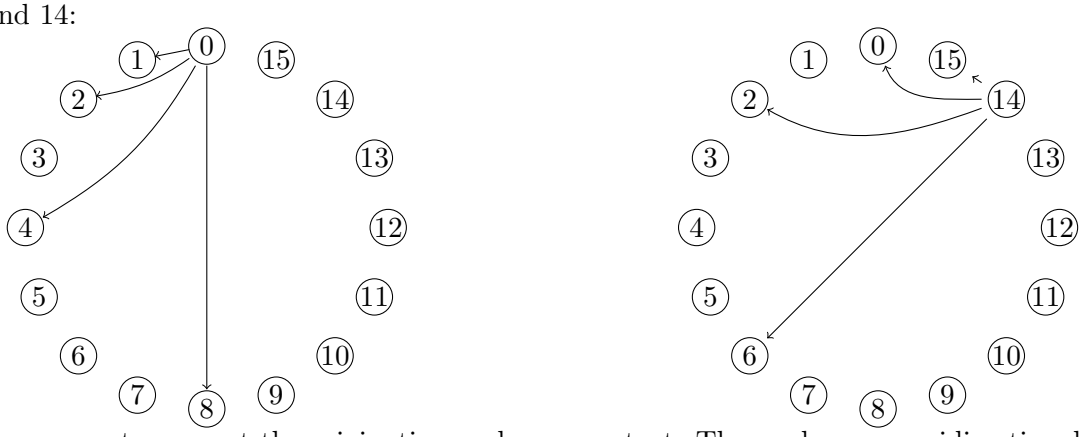
This assignment is to be done either individually or in pairs. Do not show your code to any other group and do not look at any other group's code. Do not put your code in a public directory or otherwise make it public. However, you may get help from the mentors, TAs, or the instructor. You are encouraged to use the Submitty Discussion Forum to post problems so that other students can also answer/see the answers.

In this homework you will implement an immutable distributed hash table (DHT) similar to Chord. This is a data structure that stores key-value pairs, distributed across a ring of nodes[1]. A key/value pair can be *inserted* into the DHT, or a key can be *queried* to retrieve the associated value. Since this distributed hash table is immutable, the values for each key cannot be updated after they are initially set.

You will implement your distributed hash table in either SALSA or Erlang, using the Actor Model. Each node participating in the DHT will be an actor. No additional actors are necessary.

The Network Structure

Our distributed hash table consists of 2^n nodes in a ring structure. Each node i has edges to nodes $i+1$, $i+2$, $i+4$, ... $i+2^{n-1} \mod 2^n$. Here are examples with 16 nodes ($n=4$), showing only the edges from nodes 0 and 14:



Each edge represent an agent the originating node can contact. These edges are unidirectional, meaning agent 0 can contact agent 4, but agent 4 cannot contact agent 0.

This network structure guarantees that no node is more than n hops away from any other, and that every node has only n edges. This gives us fast lookups with minimal overhead and no centralized bottlenecks. You must use the most efficient route possible; using the $i+1$ route repeatedly for $O(2^n)$ lookup is unacceptable.

[1] In the real-world, distributed hash tables store key-value pairs in several nodes for redundancy, so data is still available if the network is damaged. Your homework DHT will store each key in only one node for simplicity.

Insertion

A client can insert a key value pair into the DHT like `insert(fromNode, key, value)`. This means that the client submits their insert request to node `fromNode`. To determine which node will store the pair, the `fromNode` must hash the key. We provide a hash function from the string key to the ID of a `destinationNode`. If you are not using our provided code, the hash function is:

$$(\text{fold} + [\text{ord(c) for c in key}]) \bmod 2^n$$

Note that there may not be an edge between `fromNode` and `destinationNode`, so you will have to route the insert message to the correct node.

Nodes should only store information about keys that the hash function designates for that node. In other words, the key value pairs will be spread out amongst the nodes, and storing all key value pairs in every node, or storing all key value pairs in a single node, is an error that will result in point deductions.

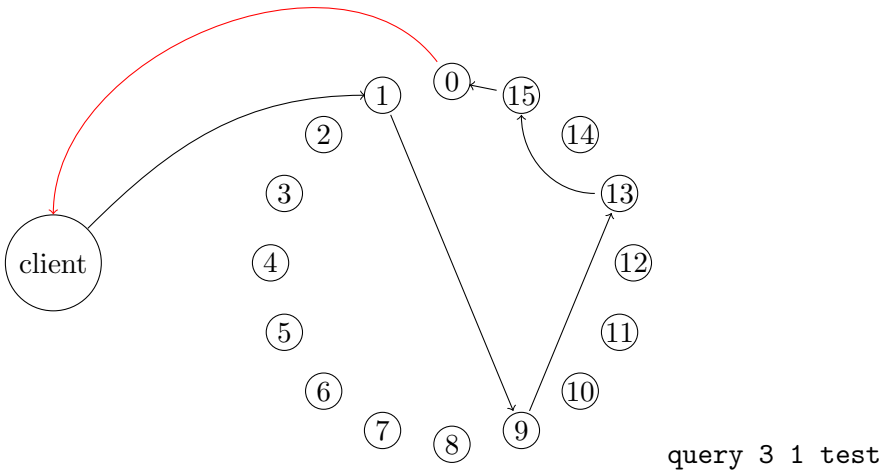
Querying

A client can request information about a key like `query(ID, fromNode, key)`. This is very similar to 'insert': The client submits their request to `fromNode`, which must hash the key to learn which node contains the corresponding value, and must forward the query to that node. When the query arrives at its destination, the node must send the corresponding value back to the client. This means you must include a reference to the client along with your query request, so the node storing the key/value pair can send the result directly to the client.

The ID field exists to help you differentiate between requests if multiple queries are sent and replied to concurrently. It is a unique integer supplied in program input.

You will never receive a query for a key that does not exist. **However:** You may receive a query for a key before the corresponding value has been inserted. If this occurs, the node where the key will be stored must *wait* until the key/value pair has been inserted, and *then* respond to the query.

Example Query



The client is an actor external to the DHT, which can send a message to any node in the distributed hash table. You may implement the client as the main actor which reads program input and starts all the other actors.

Program Input

You will receive all input from standard input. First you will receive n, the scale of your network with 2^n nodes. Then, each line will be a query or insert request. An example input is:

```
4
insert 2 test foo
insert 6 bestclass ProgLang
query 1 8 bestclass
query 2 1 bestclass
query 3 1 test
```

Note that the origin describes which agent the request should start from, and *not* which node the key/value pair should be stored in.

Program Output

Whenever a client receives the response for a `query`, it should print the result to stdout like:

```
Request 1 sent to agent 8: Value for key "bestclass" stored in node 4: "ProgLang"
Request 2 sent to agent 1: Value for key "bestclass" stored in node 4: "ProgLang"
Request 3 sent to agent 1: Value for key "test" stored in node 0: "foo"
```

Program Distribution

Your distributed hash table will be designed to run across multiple computers, with actors randomly assigned to each computer. You can simulate this on a single computer by running multiple SALSA theaters or Erlang Virtual Machines.

SALSA-specific Instructions

You must name your SALSA program's module 'pa2' for submitty to run your code correctly. Your concurrent program should be run in the following manner:

```
$ salsac pa2/*
$ salsa pa2.Main theaters.txt 127.0.0.1:3000
```

Where `theaters.txt` is a theater description file, and the last argument is the nameserver and port. See a sample theaters description file, where each line specifies a theater location.

To run your program as a distributed system, you must first run the name server and the theaters:

```
[host0:dir0]$ wwcns [port number 0]
[host1:dir1]$ wwctheater [port number 1]
[host2:dir2]$ wwctheater [port number 2]
...
```

Where wwcns and wwctheater are UNIX aliases or Windows batch scripts: See .cshrc for UNIX, and wwcns.bat wwctheater.bat for Windows. Make sure that the theaters are run where the actor behavior code is available, that is, the pa2 directory should be visible in directories: host1:dir1 and host2:dir2. Then, run the distributed program as mentioned above.

Erlang-specific Instructions

To run your program as a distributed system, you must first start an appropriate number of Erlang VMs:

```
$ erl -sname example1@localhost -setcookie foo
$ erl -sname example2@localhost -setcookie foo
$ erl -sname example3@localhost -setcookie foo -pa ./main.erl -run main -run
    init stop -noshell
```

Put `'localhost'.` in the file ".hosts.erlang", so the Erlang VM will recognize the other instances running on localhost. Now you can use the following code to dynamically find available nodes:

```
getRandomNode() ->
        World = net_adm:world(),
        Rand = rand:uniform(length(World)),
        lists:nth(Rand, World).
```

You can use the resulting node reference in calls to spawn/4 to start an actor on an arbitrary node.

Debugging Suggestions

- Start by running all agents on one node (distribution will be easier to add than you think)
- Print messages in every agent they pass through to debug routing
- Test routes that require wrapping around the ring

Provided Code

Starter code for SALSA is provided here, and starter code for Erlang is provided here. The provided code mostly parses input for you, since that is not the focus of the assignment. The provided code is also included as part of this PDF. To retrieve it, simply rename this file suffix from `.pdf` to `.zip` and decompress.

Grading

The assignment will be graded mostly on correctness, but code clarity / readability will also be a factor. If something is unclear, explain it with comments!

Submission Requirements

You will submit your code via Submitty. Include all source files needed to run your program, as well as a README file. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific features/bugs in your solution. If you are working with another student, then only one member of the team needs to submit.

Working Alone/In Pairs

You may work alone, or with one other student. To work alone, simply create a new team on Submitty with only yourself in it. To work with a partner, either invite them to your team, or join their team.