The Chord Algorithm

SDR 3.6: P2P and Advanced Distributed Algorithms

Master 2

The goal of this lab is to implement a simplified version of Chord using the SimGrid simulation framework.

★ Exercice 1: Understanding Chord

Consider the following CHORD network with 10 peers. Peer and resource keys are 6-bit identifiers.

▷ **Question 1:** How many nodes can this system host at most ?

▷ **Question 2:** Distribute resource keys K10, K23, K35, K36, K49, K55 and K63 to the network.

▷ **Question 3:** Write the finger tables of peers P12 and P36.

▷ **Question 4:** Peer P42 joins the network. Explain concisely what happen and write the finger table of P36 and P42.

▷ **Question 5:** Peer P44 leaves the network. Explain concisely what happen and write the finger table of P12.

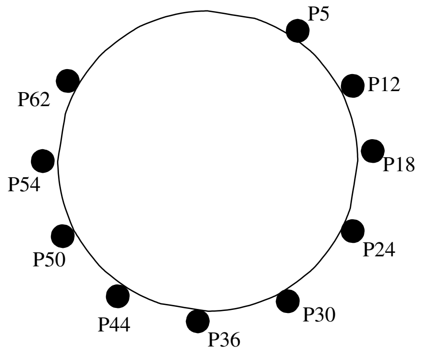

Figure 1 : Chord Network Example.

▷ **Question 6:** Look at the figure with resource keys discussed in 1. Write a routing path from peer P12 to key K55.

★ Exercice 2: First steps with SimGrid

SimGrid is a library allowing to build simulation of distributed applications. The main idea is to write the code of each agent that you want to simulate, put them on specific simulated hosts, and let the simulator tell you what would have happened if you did the same on a real platform.

▷ **Question 1:** Download the SimGrid framework from its homepage (ask google), and install it by typing : `./configure --prefix=where/you/want && make && make install`

A first toy example. To write a SimGrid code, you have to provide 3 files. The first one contains the C code of the agents you want to run, the second contains a description of the platform and the third one explains the deployment of your application, that is to say where on the platform you want to put which agent.

C code of the agents

```
1  #include <msg/msg.h>
2  XBT_LOG_NEW_DEFAULT_CATEGORY(example,"this example");
3  int alice(int argc,char *argv[]) {
4    m_task_t task = MSG_task_create("my message", 0/*comp cost*/, 10000/*comm size*/, NULL /*arbitrary data*/);
5    MSG_task_send(task,"some address");
6  }
7  int bob(int argc,char *argv[]) {
8    m_task_t task;
9    MSG_task_receive(&task,"some address");
10   INFO0("Got the message");
11 }
12 int main(int argc,char*argv[]) {
13   MSG_global_init(&argc,argv);
14   MSG_create_environment("path/to/simgrid/examples/msg/small_platform.xml"); // Change this!
15   MSG_function_register("alice", alice);
16   MSG_function_register("bob", bob);
17   MSG_launch_application("deploy.xml");
18   MSG_main();
19 }
```

deployment file

```
1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "simgrid.dtd">
3 <platform version="2">
4   <process host="Tremblay" function="alice" />
5   <process host="Fafard" function="bob" />
6 </platform>
```

Running the example

```
$ gcc example.c -lsimgrid -o example                          1
$ ./example                                                   2
[Fafard:bob:(2) 0.000000] [example/INFO] Got the message      3
```

Basically, this little first example declares two kind of processes : Alice creates a message, send it to the mailbox "some address" and quits. Bob gets the first message within the mailbox "some address", display a message, and quits. The main function of this C code (lines 12-19) initializes the SimGrid library, load a platform description file (line 14 – **you want to update this path to reflect your local conditions**), binds the "alice" string in the deployment file to the `alice` function, does the same for "bob" and `bob`, loads the deployment file (given below), and runs the simulation (line 18). The line 2 defines the default log channel. It is used on line 10 to do the equivalent of a printf, but with many information about who's speaking. Check the result in the execution on line 3. It displays the host, the process name, the process identifier and then the simulated time at which that got displayed. This is very handy to understand how your code runs.

Compiling this example can be as easy as depicted in the corresponding box, or much more complicated if you didn't install the library in the system path. Here are some hints to fix things :

– If gcc does not find the right header, add `-Ipath/to/simgrid/include` to compilation flags. Try changing `#include <msg/msg.h>` to `#include "msg/msg.h"` too if the `-I` flag does not help.
– If gcc does not find the library, you may either add `-Lpath/to/simgrid/lib` to the compilation line, or go for a static compilation by specifying `path/to/simgrid/lib/libsimgrid.a` on the compilation line (in place of `-lsimgrid`). If so, you need to add `-lm` also so that gcc finds the mathematical library (yeah, there quite a large amount of math in SimGrid). This mathematical library is found and added automatically if you go for the classical (dynamic) way of linking SimGrid.
– When running the produced binary, if the library cannot be found, you want to run this command at some point : `export LD_LIBRARY_PATH=/path/to/simgrid/lib`. You need to run it only once in a given terminal, and you may add this to your .bashrc (assuming you know what you're doing).

★ Exercice 3: Implementing Chord

The goal of this exercise is to implement a simplified version of Chord. For sake of simplification, we consider that nodes never fail or leave and that key-values pairs in the DHT are immutable. That is to say that once a key-value pair is inserted into the DHT, it cannot be deleted and the value associated with the key may not change (you should enforce this requirement). Your implementation of Chord should support dynamic insertion of nodes (but not removal), and continue to serve get and put requests simultaneously and correctly (if a value exists for a key, it must always be accessible). Keys should never reside at more than two nodes at any given time, and only one node when the ring is in a stable state.

▷ **Question 1: A Fixed Chord :** routing tables are given in the C source.

Implement a solution where the routing tables are given in the source file. You need to declare as many node kind as you will have in your deployment file (say, node5, node12, node18, node24, node30, node36, node44, node50, node54, node62 to stick to the example of the figure, but you may want to begin with less nodes).

Implement the message routing protocol. Each node (but one) needs to wait for messages on its channel (`MSG_task_receive(&task,"5") ;` and so on), and forward them according to their routing table. The remaining node (say, node12) sends a message which should be routed to another node (say, "55").

You can use `char * MSG_task_get_name(m_task_t task) ;` to retrieve the name of any given task, and assume that it is the identifier where it should be routed to.

So, you need to write the code of ten processes instead of the basic Alice and Bob in the first toy example. That should be node5, node12, node18, node24, node30, node36, node44, node50, node54 and node62. The code of each of these nodes will be very similar : wait for a message on my channel, decide if it's addressed to me or not. If not, route it to the relevant node of my routing table. That could be something like the following (with `find_successor()` being defined as in the article) :

```
void node(int id, int* routingTable) {
  char *myId=bprintf("%d",id);
  while (1) {
    m_task_t t;
    MSG_task_receive(&t, myId);
    int dest;
    sscanf(MSG_task_get_name(t),"%d",&dest);
    int next = find_successor(routingTable, dest);
    if (next==me) {
      INFO1("Got message for %d, that's for me",dest);
    } else {
      INFO2("Got message for %d, forward it to %s",dest,next);
    }
  }
}
int node12(int argc, char *argv[]){
  node(12,{18,18,18,24,30,44});
}
```

You also want to add a specific process in the simulation to do some requests. You can call it process `client`, and let it have the following code :

```
int client(int argc, char*argv[]) {
  INFO0("lookup key 55, asking to node 5");
  MSG_task_send(MSG_task_create("55", 0, 10000, NULL), "5");
  // other lookups if you want
}
```

▷ **Question 2: Adding some dynamics :** nodes can join the system.

Check the section E.1 of the Chord article. We now want to add the stabilization protocol to our code. To ensure that this gets run every once in a while, we must adapt a bit the code of the previous question. The idea is to wait at most for 5 second (for example) for a message coming to me, and run the stabilization protocol if nothing came in the meanwhile.

For that, we use `MSG_task_receive_with_timeout(m_task_t * task, const char *mailbox, double timeout) ;` instead of `MSG_task_receive()`. It returns `MSG_OK` if a message was got in the meantime, and `MSG_TIMEOUT_FAILURE` if not.

Then we should separate request messages from stabilization ones. For that we can use more complicated task names. Instead of being only the ID we are looking for, the task name can be something like "request(55)" for a request of key 55 or "notify(12)" for a notify message coming from node 12. This complicates a bit the code, but this remains doable.

Another (better) approach is to define a message structure with all the needed fields and attach it to the tasks as last argument of `MSG_task_create`. The receiver can then use `MSG_task_get_data` to retrieve the data attached to the task. This makes stuff a bit more complicated since you need to malloc memory for your structures, and add a pointer to this in the task. So, yeah, you have to do real C to go that way. But it's not that complicated either. And if you get trapped by memory errors, remember that coding in C without valgrind is a higher form of masochism...

▷ **Question 3: Removing the hard-coded routing tables**

Once the stabilization protocol is done, there is no need to hard-code the routing tables in the C code. Instead, we want to start one given node and tell it it is the first node of the swarm, and then start the other nodes by asking them to contact the guy already in the swarm. For that, we need to pass arguments to the processes. This can be done by changing the deployment file to the following :

```xml
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="2">
  <process host="Tremblay" function="node">
    <argument value="5"/><!-- First argument: my ID -->
    <!-- only argument: I'm already in the swarm -->
  </process>
  <process host="Fafard" function="node">
    <argument value="12"/><!-- First argument: my ID -->
    <argument value="5"/><!-- Second argument: ID of the guy to contact -->
  </process>
  <process host="Ginette" function="node">
    <argument value="18"/><!-- First argument: my ID -->
    <argument value="5"/><!-- Second argument: ID of the guy to contact -->
  </process>
</platform>
```

▷ **Question 4: Implementing the real interface.**

One major issue in the code we've done so far is that the requesting node does never get the answer. The message is correctly routed to the relevant node, but no answer is ever sent. To fix that, the relevant node should know who asked for the key. Again, you can (mis)use the task name for that, making your parsing code even more complicated, or you can attach pointers to structures of yours to tasks.

▷ **Question 5: Actually storing stuff.**

You should now be able to implement a function `char *find(char *key)` returning the content of the key if it was found, and NULL otherwise. This function should naturally contain a call to `MSG_task_send()` to issue the request and a call to `MSG_task_receive()` to wait for the answer. Likewise, implement `void store(char *key, char *value)`. The function `MSG_process_self_PID()`, which returns the PID of the calling process, can be handy to create private channels to get the answer.

Dictionary handling in C can quickly become a nightmare. Luckily, SimGrid implements a complete dictionary module, documented at `http://simgrid.gforge.inria.fr/doc/group__XBT__dict.html`. You can also access it by going to the SimGrid web page, and click on the XBT (toolbox) part of the picture, and then on "dictionary".

▷ **Question 6: Going further and playing with Chord.**

Your Chord implementation is now almost complete. Several paths opens in front of you. You can try changing your code to make sure that everything runs smoothly when some node fail. Then, ask SimGrid to let your node actually fail. cf. `http://simgrid.gforge.inria.fr/doc/faq.html#faq_SURF_dynamic`

You can also try writing an application using Chord for its storage purpose. Check the lecture for some examples, but a file storage naturally comes to mind. The main issue is that you don't want to store the full file under the key, but only the name of the host storing that file. If you split your file in blocks, you can even (try to) implement parallel downloads... Enjoy !