Large-Scale Distributed Systems

Practical 2: Distributed Hash Tables

1 Introduction

The goal of the next four lab sessions (29/10, 05/11, 2 x 12/11) is to build a key-routing layer and to implement the functionality of a distributed hash table on top of it. Suggested reading: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*[1].

1.1 Report and Grading

It is mandatory to deliver a report of your work to the assistant. This report will be taken into account for the final grading.

Your report should contain:

- A text report, containing the various plots you will obtain during the sessions and the associated explanations and discussions. Remember that a plot without explanation, or an assertion that is not conveyed by a plot, do not have much interest.
- Source code, properly with indentations, and with appropriate comments. Only the code of the most advanced exercise you reached for each part of the assignment is expected.

Format: Write your report in **English**. The deadline is **November, the 19th, 2:00 PM**. *Send the report to raluca.halalai@unine.ch and etienne.riviere@unine.ch.*

Use the following e-mail subject: `LSDS - assignment 2 - FIRSTNAME LASTNAME`. Only **PDF files** for the report and ".lua", ".gpi" (gnuplot), ".sh", etc. files for the code will be accepted.[2]

Grading: the assistant will grade your progress, the correctness, and the clarity of both the code and the report. The content is what matters most, but the presentation has an impact on the final grading.

[1] `http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`

[2] If you want to use another language for processing the logs, feel free to do so as long as the language is an interpreted script language: python, ruby, perl, etc.

2 The Base Chord Protocol

2.1 Constructing a Ring

Task 2.1: Implement and test the most basic configuration of Chord, in which a node does not have a finger table, but only the IDs of its direct successor and predecessor. Construct a ring of 64 nodes. The complete algorithm must provide the following:

- Nodes join the Chord network after a random time of no more than 10 seconds after starting the protocol;
- Node 1 is part of the Chord network since the beginning and it is the only node of the network that all other nodes know;
- The ID of the node is the result of hashing the concatenation of the node's IP and port (see Appendix A).

Algorithm 1 shows the pseudo code of the functions needed to construct a basic Chord ring.

Task 2.2: Evaluate the search performance of the basic Chord implementation using keys (not nodes!) as queries. Submit a large number of queries (500 per node) over the ring, using a distinct, randomly generated key for each query. The goal is to compute the average number of hops required to reach the predecessor node of the key in the ring. Plot the distribution of the number of hops required to satisfy the query. Algorithm 2 shows how to modify the function **findPredecessor(***id***)** for this task.

2.2 Adding the Finger Table

The default Chord configuration allows to perform queries over the key space in linear time. Implement an optimization to improve this result, so that queries over a ring of N nodes are performed in *O(log(N))* hops. The idea is to store at each node m pointers towards nodes further away in the ring; these are called *fingers* and are stored in a local routing (finger) table. The k-th entry of the finger table for node n points to the smallest node greater than $n + 2^{k+1}$. Note that the first finger of a node (*finger[1].node*) corresponds to the node's immediate successor in the identifiers ring.

Task 2.3: Finger table implementation. A node joining Chord must first initialize its finger table. Algorithms 3 and 4 show how to build the finger table of a node n by contacting a random node n' which is already in the network. It is assumed that n knows the address of n' (n' is the bootstrap node for n). *finger[k].start* indicates $(n + 2^{k-1}) mod 2^m, i \leq k \leq m$.

Task 2.4: When a new node n joins, it can become the successor and/or the finger of other nodes in the network. To account for this change, fingers and successors of current nodes require to be updated[3]. Algorithm 3 also illustrates this procedure.
Task 2.5: Performance Comparison. Evaluate how the presence of the finger table impacts the performance of queries over the ring. Re-run the experiment from Task 2.2 using Chord

[3]The number of nodes that actually require this update is on average *O(log(N))*. See original Chord paper for a formal proof.

Algorithm 1: Algorithm for Chord with no finger tables

Variables

- n: the node itself;
- $n.id$: the node's ID;
- m: size of ID in bits;
- *successor*: the closest node with bigger ID, initially `nil`;
- *predecessor*: the closest node with smaller ID, initially `nil`;

find_predecessor(id**)**

- $n' = n$; `// you start the search with yourself`
- n'*_successor* = *successor*;
- `/* important:` n'*_successor* `may have smaller ID than` n' `*/`
- **while** $id \notin (n'.id,\ n'\_successor.id]$ **do**
  - $n' = n'$*_successor*;
  - n'*_successor* = invoke get_successor() on node n';
- `return` n';

find_successor(id**)**

- n' = find_predecessor(id);
- n'*_successor* = invoke get_successor() on node n';
- `return` n'*_successor*;

init_neighbors(n'**)**

- `/* find (through node` n'`) the successor of your ID */`
- *successor* = invoke find_successor($(n.id + 1) mod (2^m)$) on node n';
- *predecessor* = invoke get_predecessor() on node *successor*;
- `/* update your neighbors */`
- invoke set_predecessor(n) on node *successor*;
- invoke set_successor(n) on node *predecessor*;

join(n'**)**

- **if** n' **then**
  - init_neighbors(n');
- **else**
  - *successor* = n;
  - *predecessor* = n;

Algorithm 2: Changes to the algorithm for calculating distance in Chord

```
find_predecessor(id)
    n' = n;
    n'_successor = successor;
    i = 0;
    while id ∉ (n'.id, n'_successor.id] do
        n' = n'_successor;
        n'_successor = invoke get_successor() on node n';
        i = i + 1;
    return n', i;
```

with the finger table, and compare the new results with the previously obtained ones.

3 The Fault-Tolerant Chord Protocol

Extend Chord to improve its resilience to faults, to support concurrent joins and departures of nodes. Test the effectiveness of this extension by means of a churn trace (that is, a description along a timeline of the availability of the nodes in the ring).

3.1 Churn Trace

The churn trace is a text file which describes the availability of the nodes along the experiment's duration. Download dht.churn from ILIAS. The trace describes the following behavior:

- in the first 1m40s, 64 nodes join the network
- after 2 minutes from the beginning of the experiment, 14 random nodes are removed forever from the network
- after other 30 seconds, 30% of the currently alive nodes are removed forever from the ring
- 3 minutes after the start, 20% new nodes (of the current network size) are injected in the network
- finally, after 5 minutes, all nodes are removed.

Figure 1 shows how to upload the churn trace via SplayWeb.

Task 3.1: Using the pseudo code given in Algorithm 5, extend the basic protocol to support stabilization. The aim is to update the successor list and the finger tables separately. Our priority is to periodically update the successor list which is sufficient to guarantee the correctness of the ring; periodically updating the finger tables allows for faster lookups, without slowing down the process of learning the latest state of the ring.

Algorithm 3: Changes in initialization for Chord with finger table

```
Variables
    finger: table of known nodes;
    define successor = finger[1].node;
/* init_neighbors(n') is replaced by these three functions:                    */
init_finger_table(n')
    finger[1].node = invoke find_successor(finger[1].start) on node n';
    successor = finger[1].node;
    predecessor = invoke get_predecessor() on node successor;
    for i = 1 to m-1 do
        if finger[i + 1].start ∈ [n, finger[i].node) then
            finger[i + 1].node = finger[i].node;
        else
            finger[i + 1].node = invoke find_successor(finger[i + 1].start) on n';
update_finger_table(s, i)
    if finger[i].start ≠ finger[i].node & s ∈ [finger[i].start, finger[i].node) then
        finger[i].node = s;
        p = predecessor;
        invoke update_finger_table(s, i) in node p;
update_others()
    invoke set_predecessor(n) on node successor;
    for i = 1 to m do
        p = find_predecessor((n + 1 − 2^(i−1)) mod 2^m);
        invoke update_finger_table(n, i) on node p;
/* join() changes to                                                           */
join(n')
    if n' then
        init_finger_table(n');
        predecessor = invoke get_predecessor() on node successor;
        update_others();
    else
        for i = 1 to m do
            finger[i].node = n;
        predecessor = n;
```

Algorithm 4: Changes in routing for Chord with finger table

```
/* find_predecessor(id) is replaced by these two functions:          */
closest_preceding_finger(id)
    for i = m down to 1 do
        if finger[i].node ∈ (n, id) then
            return finger[i].node;
            ;
find_predecessor(id)
    n' = n;
    n'_successor = successor;
    while id ∉ (n'.id, n'_successor.id] do
        n' = invoke closest_preceding_finger(id) on node n';
        n'_successor = invoke get_successor() on node n';
    return n';
```

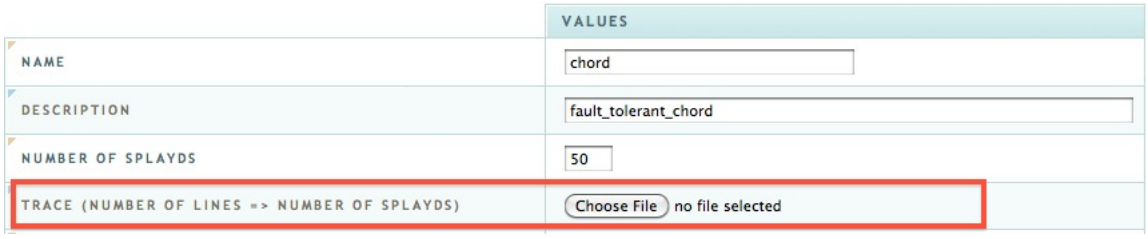

Figure 1: How to upload a churn trace.

Algorithm 5: Algorithm for Stabilization

```
/* These four functions replace Algorithm 3                          */
join(n')
    predecessor = nil;
    successor = invoke find_successor(n) on node n';
stabilize()                    // this function must be periodically executed
    x = invoke get_predecessor() on node successor;
    if x ∈ (n, successor) then
        successor = x;
    invoke notify(n) on node successor;
notify(n')
    if predecessor = nil or n' ∈ (predecessor, n) then
        predecessor = n';
fix_fingers()                  // this function must be periodically executed
    i = random(2, m);
    finger[i].node = find_successor(finger[i].start);
```

Task 3.2: Plot the stale references of the finger tables. A *stale reference* is a finger table entry which is not valid, *i.e.*, it points to a node not available anymore due to churn. Plot the average number of stale references with respect to the experiment lifetime.

Task 3.3: Produce a plot to show how *fast* the protocol recovers from node failures/departures. Use different values for the periodic tasks and explain how they affect the performance.

Task 3.4: Search performance evaluation under churn. Re-run the same experiment as in Task 2.2 and show how it impacts the success ratio of the queries. When there is no churn, the success ratio must be 100%. Compare your results with the node availability induced by the churn trace.

Task 3.5: *Optional* Study the behavior of the fault-resilient protocol under the occurrence of a catastrophic event, as described by the churn trace: dht-dramatic.churn. In this scenario, 50% of the population dies after 3 minutes during the experiment. The remaining nodes last until the 6th minute.

A Hashing function

The `SHA-1` hash function is provided by the `crypto` Lua library. The following code snippet illustrates how to use it.

```
require"splay.base"
crypto = require"crypto"
m = 32
function compute_hash(o)
    return string.sub(crypto.evp.new("sha1"):digest(o), 1, m / 4)
end
print("IP: 192.168.0.24 and port: 18753 hash to :")
print(compute_hash("192.168.0.24"..":".."18753"))
#c055177e
```