Master of Science Thesis in Software Engineering and Management

Testing implementations of Distributed Hash Tables

Vinh Trương

Göteborg, Sweden 2007

IT University of Göteborg

CHALMERS | GÖTEBORG UNIVERSITY

REPORT NO. 2007/41

Testing implementations of Distributed Hash Tables

VINH TRƯƠNG

Department of Applied Information Technology
IT UNIVERSITY OF GÖTEBORG
GÖTEBORG UNIVERSITY AND CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Testing implementations of Distributed Hash Tables
VINH N. X. TRƯƠNG

Report no. 2007:41
ISSN: 1651-4769
Department of Applied Information Technology
IT University of Göteborg
Göteborg University and Chalmers University of Technology
P.O. Box 8718
SE – 40275 Göteborg
Sweden
Telephone +46 31 - 772 4895

Göteborg, Sweden 2007

Testing implementations of Distributed Hash Tables

VINH N. X. TRƯƠNG

Department of Applied Information Technology
IT University of Göteborg
Göteborg University and Chalmers University of Technology

Supervisor: Thomas Arts

SUMMARY

A lot of research about peer-to-peer systems, today, has been focusing on designing better structured peer-to-peer overlay networks or Distributed Hash Tables, which are simply called DHTs. To our knowledge, not many papers, however, have been published about testing implementations of them. This thesis presents an attempt to test an implementation of Chord, one of the well known DHTs, with a property-based testing method. We propose an abstract state machine as a way to model a DHT, and as a correctness property that its implementation is supposed to satisfy. As a case study, we test the Chord implementation with a property-based random testing tool, showing some faults in the implementation, and suggesting some modifications in its original algorithm.

Keywords: Distributed Hash Table, property-based random testing, abstract state machine

Acknowledgments

I would like to thank my supervisor, Thomas Arts, for his enduring support and guidance. From when I started learning the Erlang programming language in his Test and Verification course until today, he has been a constant source of encouragement. He has inspired me with his positive outlook and confidence in my research. In particular, this thesis benefits greatly from his insights, suggestions and patient revisions.

I must thank all my classmates in the Software Engineering and Management Master program for being good companies. They have persistently stimulated my research by posing useful questions and suggestions. I wish them every success in their future.

I also thank all the program managers, teachers, supervisors and coordinators in this program, who have been working hard to keep the program going and improving.

I am also grateful to Swedish Institute, not only for providing the funding which allows me to pursue this education but also for giving me the opportunity to attend seminars, excursions and meet so many interesting people.

And most of all, I would like to thank my family for their understanding, endless support and encouragement; my parents, my wife and my daughter, who have always been there waiting for me. I will soon be home.

Vinh Trương
Göteborg, April 2007

Contents

Figures

Tables

Chapter 1. **Introduction**

This chapter presents an introduction to our thesis. We start with a motivation behind this research, following a summary of its objectives and contributions. We then mention some related works and outline the structure of this thesis document.

1.1. Motivation

The popularity of file-sharing and multimedia over IP services has recently motivated intensive research into peer-to-peer systems, especially in the area of structured peer-to-peer overlay networks or Distributed Hash Tables, which are simply called DHTs. In the literature that describes those systems, the correctness proofs are usually stretched at a high level of abstraction, and tends to use simulations to evaluate both the designs and their implementations.

Overall, that method means to test the correctness of the entire system viewed as a black-box. Commonly, it leaves out the corner cases where bugs are typically hidden. And even when bugs are detected, it is very difficult to trace out the actual causes.

With this thesis, we propose a new approach to test the implementations of DHTs by using a property-based random testing method. Basically, our approach is a combination of three techniques:

- modeling a DHTs with an abstract state machine; defining a state as a collection of nodes and their relationships;
- using a property-based random testing tool to generate random test cases from the model; simulating its state machine and the dynamic environment that the DHT can operate in;
- executing the test cases on the DHT implementation; checking real states against its modeled states, and real nodes' relationships against their modeled ones; automatically shrinking the failing test cases.

As a case study, we apply this method to test a Chord [1] (one of the well known DHTs) implementation using Erlang QuickCheck [7] [10] [11].

1.2. Objectives and contributions

Formally, the main question of our research is:

– How can the implementations of DHTs be effectively tested?

To answer this main question, we try answering these two immediate ones: (1) What to test and (2) How to test. The first question is to find the appropriate cases to verify. In this research, it is to find the properties that a DHT system is supposed to satisfy. The most challenging problem in this stage is to find the appropriate properties, which together capture the main functionality of the DHT.

By doing so, instead of proving the correctness of the implementation itself, we can prove the correctness of these specified properties instead. The second immediate question is to correctly set up a simulation and carry out test cases on it. Two things that need to be considered are the programming language we use to implement the system and the tool we use to test it.

These two questions are designed to achieve the main research objective:

– Developing a novel testing method for testing implementations of DHTs

Altogether, they result in a research contribution by means of a case study on how DHTs can effectively been tested. It combines research from two areas; viz. DHT and property based testing.

Furthermore, as a result of testing the Chord implementation, we contribute with a new version of it and a modified version of Chord algorithm.

1.3. Related works

The attempt to use a property-based testing method to test implementations of formally verified algorithms and protocols should be seen as one among several similar projects. Arts et al. [8] [9] presented a case study of testing a formally-verified leader election algorithm. The leader election algorithm is an algorithm that helps elect a single node as a leader among many competing ones. Although it is not a peer-to-peer algorithm, leader election is a solution for distributed networks, which involves distributed nodes and concurrent processes. In their research, the leader election algorithm has been tested with the same property-based random testing tool that we are using. The research is related to ours in the sense that we share experience and findings of testing two different formally verified algorithms. In [11], the same testing tool is used to develop and test an industrial implementation of the Megaco (ITU-T H.248) protocol, a component interfacing to Ericsson's media proxy.

An attempt to model a DHT at an abstract level could have been seen in [13]. The authors proposed an abstract model for structured peer-to-peer network with ring topology. Different from ours, their choice of modeling formalism is π-calculus. As a case study, they use the π-calculus to model both the specification and the implementation of Chord system.

1.4. Overview

In Chapter 1, a motivation behind our research, a summary of its objectives and contributions together with some related works have been presented.

Chapter 2 gives a short technical overview of Distributed Hash Tables, Erlang programming language and QuickCheck.

In Chapter 3, we introduce our model of Chord protocol. There, we propose a way to separate DHT registration from DHT logic, and explain how they are as individual and as together modeled with finite state machines. We then specify what properties should be hold in those models.

Chapter 4 presents our first Chord implementation. The implementation is based on the original Chord paper by Stoica et al. [1]. We explain how the algorithm presented in that paper is implemented in Erlang, and how we fill the gap between the implementation and its original algorithm by replacing the Remote Procedure Calls (RPCs) by the for Erlang more natural asynchronous message.

Chapter 5 is for a case study of testing the Chord implementation presented in Chapter 4 against the model that we propose in Chapter 3, using the tool QuickCheck introduced in Chapter 2. In more details, we present how to use QuickCheck as a scheduler to simulate a Chord system and how to write properties, generate and execute test cases.

In Chapter 6, we show several serious faults in the Chord implementation. Together with an analysis, we suggest some modifications in the original Chord algorithm.

Next, in Chapter 7, we discuss the results in a broader context, where we include the cases of other popular DHTs. Some directions for future research are also mentioned.

We conclude this thesis with a summary of our works in Chapter 8.

Chapter 2. **Background**

This chapter presents the background of our research. That includes the basics of Distributed Hash Tables, especially Chord, and a short overview of Erlang and QuickCheck, the programming language and the testing tool we use.

More information about QuickCheck could be found in [7], [10] and [11]. For fuller treatment of Erlang, the reader is referred to the Erlang book [14] and the Erlang website [34]. The formal descriptions of DHTs are provided by various normative and informative references in the References section.

2.1. Distributed Hash Tables

Peer-to-peer systems can be broadly classified into unstructured (such as Kazaa and Gnutella with no structure of how the users store the data) and structured (such as those using DHTs). A DHT is a distributed data structure that efficiently maps “keys” to “values”. Users of a DHT must use keys to store and retrieve the corresponding values. In a DHT, nodes maintain information of a small number of other nodes, forming an overlay network. To implement a DHT, the underlying algorithm must be able to determine which user is responsible to the data associated with a given key. Different from unstructured peer-to-peer networks, messages in DHT networks are routed in some special schema, instead of flooding them expensively.

There have recently been many proposed networks that implements DHTs, including Chord, CAN [2], Pastry [3], Tapestry [4], Viceroy [5] and Kademlia [6]. These DHTs differ to each other in the way they implement the routing geometry. For instance, Chord routes messages along a ring, CAN routes along a hypercube, while Pastry and Tapestry uses a ring-like geometry in addition to a tree. Each DHT has its own advantages in terms of the way it handles the situations of concurrent arrivals and departures. Table 2.1 summarizes these well-known DHTs and their routing geometries.

DHT	Geometry	Description
Chord	Ring	Nodes organize into a logical ring ordered by IDs
CAN	Hyper-cube	Nodes organized into a d-dimensional torus
eCAN	Torus	Messages are routed in d-dimensional paths
Pastry	Hybrid	Nodes organize into a tree, but leaves form a ring
Tapestry, TOPLUS	Tree	Pure tree
D2B,Koorde	de Bruijn	Nodes organize into a de Bruijn graph
Viceroy	Butterfly	Nodes organize into a butterfly graph

Table 2.1: DHT geometries

Table 2.2 (Source: Araujo and Rodrigues [15]) summarizes these DHTs with their performance and flexibility complexities.

P2P Systems	Node degrees	Network diameter	Node congestion	Optimal path	Neighbor selection
Small-worlds	O(1)	$O(\log^2 n)$	$O((\log^2 n)/n)$		
Chord	O(log *n*)	O(log *n*)	O((log *n*)/*n*)	O(log *n*)	$n^{logn/2}$
CAN	O(*d*)	$O(dn^{1/d})$	$O(dn^{1/d-1})$	O(log *n*)	1
Pastry	O(log *n*)	O(log *n*)	O((log *n*)/*n*)	1	$n^{logn/2}$
Tapestry	O(log *n*)	O(log *n*)	O((log *n*)/*n*)	1	$n^{logn/2}$
D2B	O(1)	O(log *n*)	O((log *n*)/*n*)	1	$n^{logn/2}$
Viceroy	O(1)	O(log *n*)	O((log *n*)/*n*)	1	1
Koorde cfg. 1	O(1)	O(log *n*)	O((log *n*)/*n*)	1	1
Koorde cfg. 2	O(log *n*)	O((log *n*)/log log *n*)	O((log *n*)/(*n* log log *n*))		

Table 2.2: DHT performance and flexibility complexities

Chord is one of those well-known DHTs. As other networks that implement DHTs, Chord uses keys to store and retrieve data. In Chord, ID's are assigned to both data and nodes. The node responsible for key k is called its successor, defined as the node whose ID most closely follows k.

Chord differs from other DHTs in the way that all of its entities are organized in a ring, and the routing procedure is one-dimensional. During registration, a node is assigned a unique ID chosen by hashing its address (e.g. IP address). Depending on IDs, nodes are logically located at the correct positions.

During operation, a node needs to maintain a record of log N other nodes, where N is the total number of them. That record is called the successor list. Those other nodes are called the node's successors (the immediate successor is one of them). By keeping information of log N successive neighbors, a node, with high probability, can be secure to have some successor to communicate to in case its immediate successor fails.

A Chord node also maintains a finger table of log N entries as its routing table. The i^{th} entry in the finger table contains the node that succeeds by at least 2^{i-1} other nodes on the ID ring. The role of the finger tables is to reduce the average number of hops when routing messages. More specifically, it helps reduce the lookup time from O(N) of sequential routing to O(logN).

Figure 2.1a shows a network with 4 nodes and an ID space of 2^3. Each node has 3 entries in its successor list and the same number in its finger table.

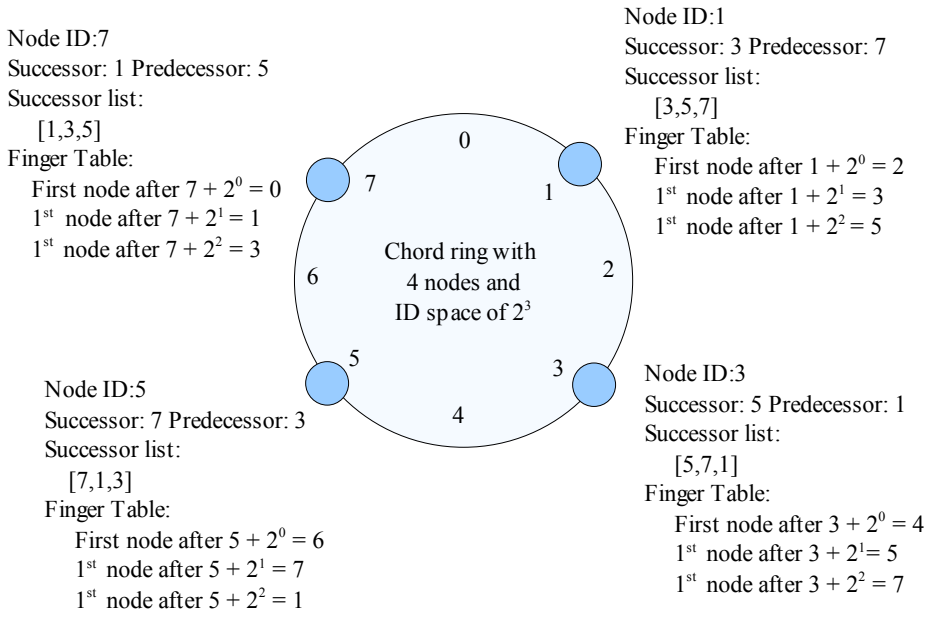


Figure 2.1a: A sample Chord network: 4 nodes and 2^3 IDs

Chord deals with the issues of concurrent arrivals and voluntary departures by implementing an algorithm called stabilization. Every node, during operation, periodically sends out requests to ask for their successors' predecessor, giving a chance for a node to check and update its correct neighbors. Since every node runs stabilization, they can detect new arrivals and departures, and reorganize accordingly.

Figure 2.1b shows the use of stabilization on arrival when node 201 joins the network. First, it asks the gate (12) to find its immediate successor. (b) Successor 328 is found. By that time, the rest of the network has not been aware of the new node yet. (c) 165 runs stabilization and detects node 201 as its successor. In turn, node 328 runs stabilization and detects 201 as its predecessor.

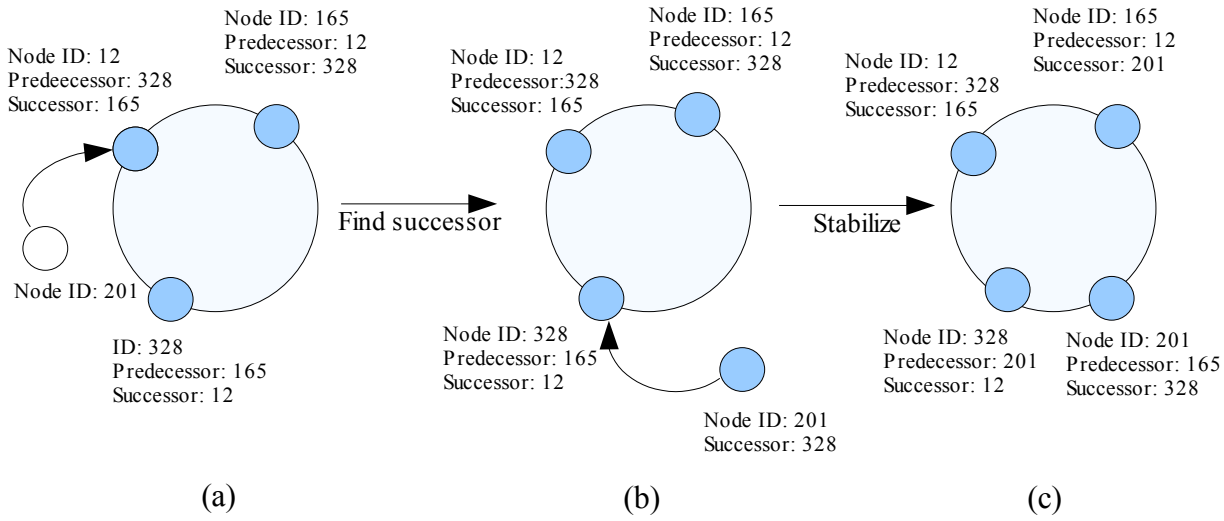


Figure 2.1b: Stabilization on arrival

2.2. Erlang

Erlang is a high-level and declarative programming language with support for concurrent and distributed programming. The language has been developed at Ericsson and is typically used to implement concurrent, real-time distributed systems (e.g. telecommunications).

In Erlang terminology, a distributed system consists of nodes, which themselves contain multiple processes. Each process has its own memory and an incoming mailbox. A process is created with an Erlang primitive *spawn*. This function returns the process identifier of the new process. Processes communicate with each other by sending messages. This is done by the send and receive primitives: "!" and "*receive...end*" respectively.

Erlang deals with the issues of process failures by implementing reactive detection mechanisms: process linking and monitoring. A process can obtain a bidirectional *link* or a unidirectional *monitor* on another process. If another process fails, the monitored process receives a message informing about the failure. In a dynamic and distributed environment, the reactive detection mechanisms help the system actively and quickly recover from failures.

Figure 2.2 shows an example of starting two Erlang processes, one is monitored by another.

```
-module(left).

-import(erlang,[monitor/2]).
-export([left/0,init/0]).

left() ->
   register(left,spawn(?MODULE,init,[])).

init() ->
   process_flag(trap_exit,true),
   Right = monitor(process,right),
   loop(Right).

loop(Ref) ->
   receive
      {'DOWN',Ref,process,_,_} ->
          do_some_action()
   end.
```

```
-module(right).

-import(erlang,[monitor/2]).
-export([right/0,init/0]).

right() ->
   register(right,spawn(?MODULE,init,[])).

init() ->
   process_flag(trap_exit,true),
   Left = monitor(process,left),
   loop(Left).

loop(Ref) ->
   receive
      {'DOWN',Ref,process,_,_} ->
          do_some_action()
   end.
```

Figure 2.2: Sample Erlang programs

In the above example, the function *left* is used to *spawn* a process, and *register* that process with a local name **left**. The function is *export*ed, so that it can be called by another *module*. Similarly, *right* is used to spawn and register another process: **right**. The built-in function *monitor* is *import*ed and used to monitor one process by another.

When **left** fails, a message {'DOWN',_,process,_,_} will be sent to **right**, initiating it to take a proper action.

The flag of *trap_exit* is set to true to ensure that a process will never be automatically terminated when receiving any exit signal from other processes.

2.3. QuickCheck

Erlang QuickCheck is a property-based tool for random testing. Users write properties of a system based on its specification and let the tool generate and execute random test cases. An advantage of QuickCheck is to reduce the number of test cases. Another advantage is that it covers corner cases. Based on the earlier work of QuickCheck tool for Haskell [12], the Erlang adaptation version extends with the feature to shrink failing test cases automatically. It helps to find the actual cause of a fault, especially when the test cases are very large.

Regarding to large and complex systems, QuickCheck has been used in several protocol testing case studies, including testing an implementation of leader election algorithm, and the Megaco protocol in an Ericsson's media proxy. A new version of QuickCheck introduces the functions to test operations with side-effects, which are specified by an abstract state machine. To simplify the test cases by separating test generation from test execution, furthermore, the tool introduces the concepts of symbolic calls and dynamic variables.

In this section, we explain some basic QuickCheck concepts and primitives by showing an example of testing a simple system – an electric lamp. The state machine of this system can be drawn as in Figure 2.3a.

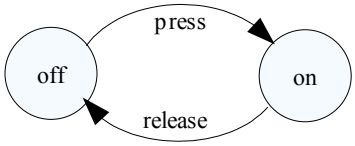


Figure 2.3a: State machine of an electric lamp

In QuickCheck, we use *command* to generate random sequences of **press**'s and **release**'s. When a command press is called, it checks the *precondition* that the current state of the system is OFF, and checks the *postcondition* that the new state of the system is ON.

```
-module(lamp).

-include("eqc.hrl").
-include("eqc_statem.hrl").

-export([command/1, next_state/3,
         precondition/2, postcondition/3,
         initial_state/0, prop_lamp/0,
         press/0, release/0]).

-behaviour(eqc_statem).

prop_lamp() ->
  ?FORALL(Cmds,commands(lamp,initial_state()),
    begin
      {_,_,R} = run_commands(?MODULE,Cmds),
      R == ok
    end
  ).

initial_state() -> on.

command(S) ->
  frequency(
    [{1,stop}] ++
    [{10,{call,?MODULE,press,[]}} || S == off] ++
    [{10,{call,?MODULE,release,[]}} || S == on]).

press() -> on.
release() -> off.

precondition(S,{call,_,press,_}) -> S == off;
precondition(S,{call,_,release,_}) -> S == on.

postcondition(_,{call,_,press,_},R) -> R == on;
postcondition(_,{call,_,release,_},R) -> R == off.

next_state(_,_,{call,_,press,_}) -> on;
next_state(_,_,{call,_,release,_}) -> off.
```

Figure 2.3b: A sample QuickCheck specification

?FORALL is an Erlang macro, which binds the first argument to a value in the set. *?MODULE* is another Erlang macro, which stands for the name of the program. The function *run_commands* runs a list of commands, which are generated by *command*. The weight associated with each command is the probability that that command is chosen in the *frequency* function. The function *next_state* is used to change the state of the modeled state machine according to the command executed.

```
1> eqc:quickcheck(lamp:prop_lamp()).
Starting eqc version 1.07
.................................................................................
.................
OK, passed 100 tests
true
```

Figure 2.3c: A sample QuickCheck output

Figure 2.3c shows the output when we call to test the property *prop_lamp*. QuickCheck tests the property in 100 random cases, equivalent to 100 sequences of *press*'s and *release*'s. The result is true, as above, if all test cases succeeded.

Chapter 3. **Modeling**

In this chapter, we propose finite state machines (FSMs) to model DHTs. We first discuss the system-level view of DHTs. We then introduce a simple model based on DHT registration. A more complete model based DHT logic is next presented.

Peer-to-peer systems in general and DHTs in particular are inherently dynamic and have infinite-state behavior. That is the main challenge to model them at a high level of abstraction. The underlying philosophy of our model, therefore, is to view an infinite state machine as a collection of finite state machines and view a multi-dimensional model as several two-dimensional ones.

Simply speaking, we suggest viewing peer-to-peer systems from their top level, where we take into consideration the whole collection of nodes in addition to the whole collection of their relationships. This view is different from the one-single-node view used for server-client architecture (also distributed systems). In a server-client system, we typically construct a finite state machine of the server, and observe the operation of the server-client system based on the finite state machine in that server side only.

With a top-level view, as mentioned, we define a state of a state machine as a collection of nodes and relationships. The relationships could be a node as the successor of another as in Chord or a node in an adjacent zone as in CAN. The machine changes its state when the collection of nodes or the collection of their relationships changes. In Chord, such actions could be an action when a new node joins, or when a node performs its maintenance activities.

Action	**Description**	**New State**
start	a new node starts	A state with one node
join	a new node joins	A state with an additional node
stop	a node leaves	A state with one node less
stabilize	a node runs its periodic stabilization	The predecessor of the node's successor is supposed to be updated
update_successors	a node runs its periodic update of successor list	An additional entry is inserted into the node's successor list
update_fingers	a node runs its periodic update of finger table	An additional entry is inserted into the node's finger table

Table 3: Chord actions

Table 3 summarizes all the actions that could change state of a Chord system. The actions consist of two observable events, the first to trigger the action and the second to acknowledge that the action has been performed. The actions, therefore, have to be observed as a pair of events. We also note that an *update_successors* could be considered as a *stabilize* if we extend the *stabilize* so that it also maintains successor lists.

3.1. Registration

Based on the summary of DHT actions, our first step to model a DHT is to separate the user registration from its overlay logic. In other words, we separate the collection of nodes from the collections of relationships. At this high level of abstraction, we leave out the relationships among nodes, and assume that the recovery activities are continuously running in the background. The model of Chord at this level, therefore, is concerned with three actions: *start*, *join* and *stop*. Those are the functions used to register and unregister a user, and by themselves, change the collection of nodes.

Figure 3.1 shows a finite state machine of a Chord system with three nodes. These three are initially all inactive. For simplicity purposes, we do not include that initial state in the figure. The reader, however, should be aware that there are totally 2^3 states in this FSM.

Starting with a network of two nodes 1 and 2 active, an action *join(3)* will change the machine to a state when all the nodes are active. Similarly, an action *stop(2)* will change the machine to a state when node 2 is inactive.

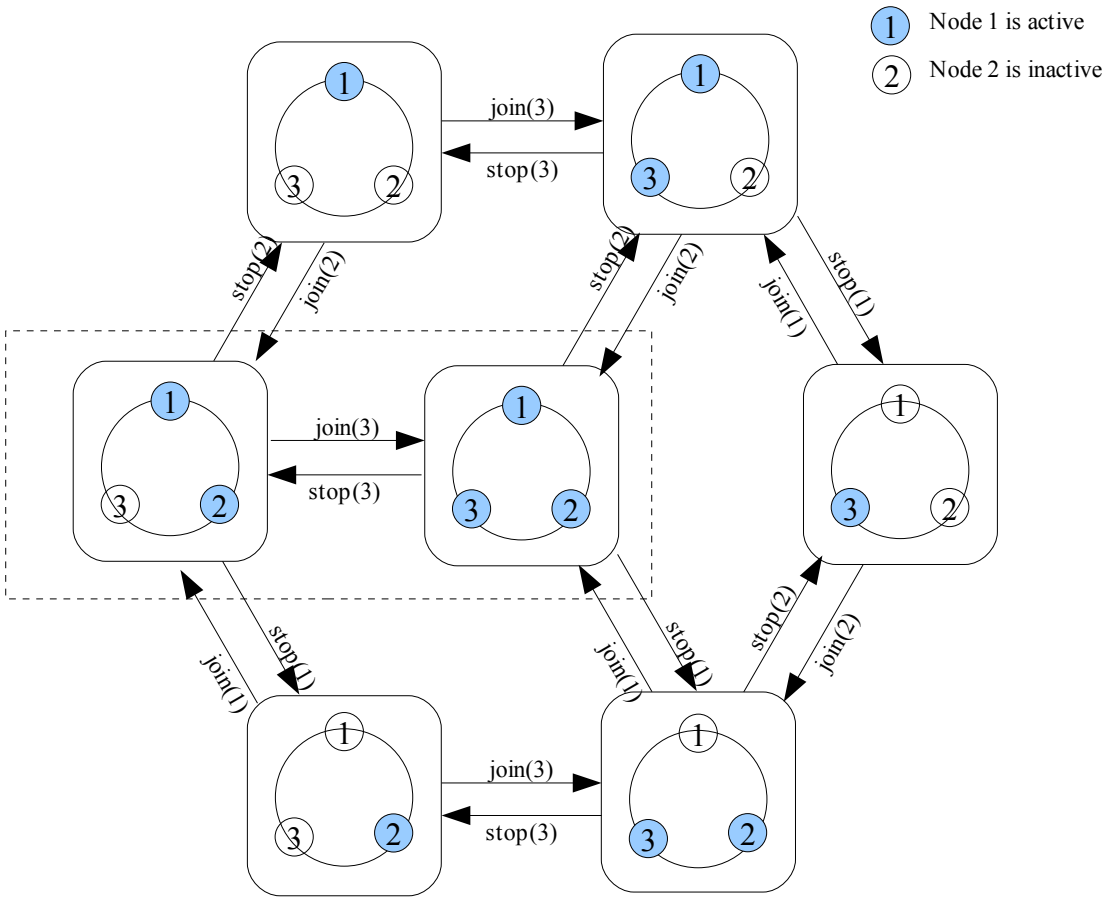


Figure 3.1: Model of Chord registration

As mentioned, this registration FSM is viewed from the top level, with the assumption that the maintenance and recovery activities are continuously running in the background. It means the *join* that we consider here is actually *join* + *stabilize* or *join* + time for stabilization if considered from a lower level.

The registration FSM would be enough for black-box testing, as far as we add a proper stabilization time to each interface function. We, ourselves, have tried this to test the Chord implementation in our P2P SIP project [37]. The problem, however, was that it is difficult to troubleshoot errors, especially when they are from maintenance and recovery algorithms. We could not observe what is really happening during that stabilization time either. A more complete model, therefore, is needed to model the system in more details by removing the assumption of stabilization time and background activities.

3.2. Logic

The logic of Chord includes functions to maintain DHT peers information: predecessor, successor list and finger table. Among six actions listed in Table 3, three of them are considered as maintenance and recovery actions and are of the Chord logic. They are *stabilize*, *update successor list* and *update finger table* actions.

In our model of Chord logic, we consider *stabilize* and *update_successor*s as two separate actions. The model, however, is still applicable to the case when *stabilize* is modified to also maintain successor list.

Figure 3.2a shows a finite state machine of a network of three nodes. This corresponds to the dotted rectangle portion of the finite state machine shown in Figure 3.1. The difference is that we now include *stabilize(2)* after *join(3)* and *update_successors(2)* after *stop(3)*; in addition to a hyper-state of three active nodes. We further show that hyper-state portion in Figure 3.2b.

In this model, a state is defined as a collection of three nodes 1, 2 and 3. Each has a predecessor and a successor list. For example,

1,2,{2,_}

means that node 1 has 2 as its predecessor and as the first entry in its successor list. If the ID space size of this network is 2^2, the size of the successor list is 2. The second entry of the successor list is currently undefined or unknown.

The action *join(3,1)* leads to a successor update at node 3. At the predecessor of 3, *stabilize(2)* will then update the successor for that node.

At the other end, the action *stop(3)* leads the FSM to a state when 3 do not have any predecessor and successors (inactive as defined in the previous registration model). Node 2 detects the failure of its immediate successor, so it replaces that with the next entry in the successor list. Node 3 as the second entry in the successor list of node 1 will act as the immediate successor when *update_successors(1)* is called.

The model in Figure 3.2a, specified by a FSM, shows these transactions in more details.

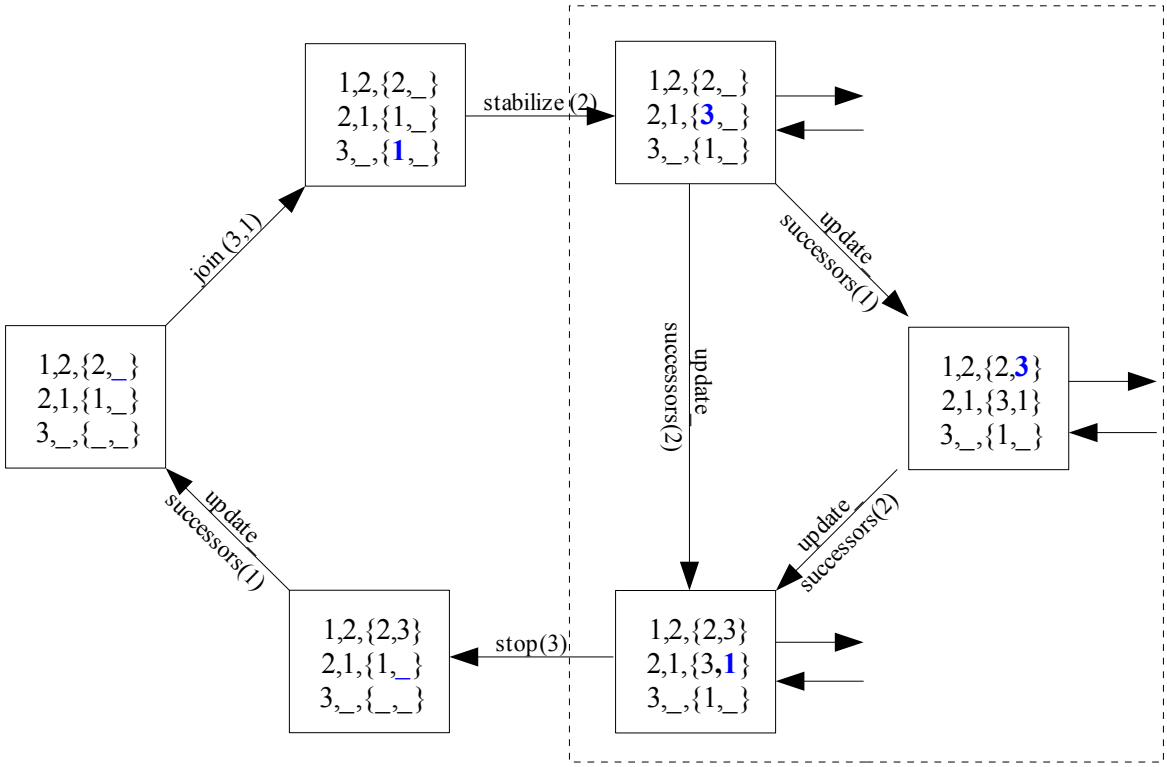


Figure 3.2a: Model of Chord logic in the face of node joins and leaves

The FSM portion bordered by the dotted rectangle is further explained in Figure 3.2b. In Figure 3.2b, for simplicity purposes, we only consider *stabilize* actions. We assume that they are the *modified stabilize* which maintain successor lists. It means an action *stabilize* updates not only the immediate successor but also all other entries in the successor lists.

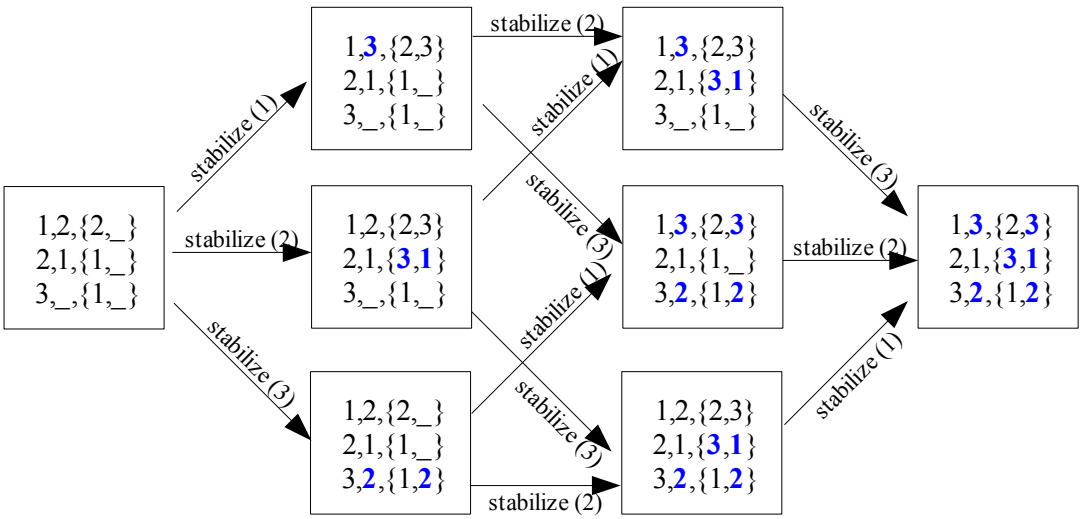


Figure 3.2b: Model of Chord logic without any node join and leave

In Figure 3.2b, there are finitely 8 states starting from the time when node 3 has just joined the system until the time when the system becomes complete stable. Together, as a FSM, it complements to the FSM shown in Figure 3.2a. Overall, we concatenate two FSMs (Figure 3.2a and Figure 3.2b) to form one FSM and embed it into the top-system-level FSM shown in Figure 3.1. A complete model of Chord, by that way, has been constructed.

The **blue** numbers in the model indicate the change of states, and are intended to be used for testing. For example, *stabilize(1)* is tested by simply checking the predecessor '3' of node 1, leaving out other unchanged information.

State could also be extended to include figure table records after successor lists as shown in Figure 3.2c.

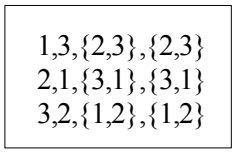


Figure 3.2c: State with finger table record

Chapter 4. **Chord implementation**

In this chapter, we present the first version of our Chord implementation. The implementation is loosely based on the Chord protocol described by Stoica et al. To ease the explanation, some fragments of the code are pasted into here. The modified version of it could also be found in Appendix B.

Overall, Chord is implemented as three separate modules. The user registration module is responsible for registering nodes in the system. It includes functions to *start* the network and *join* new nodes into it. The data location module is used to locate users. It includes functions to *lookup* responsible nodes. The logic module contains overlay maintenance functions, including *stabilize*, *update_successors* and *update_fingers*.

4.1. User registration

A new node is considered to be successfully registered if it is placed at a correct position in the network. In Chord, that means correct immediate successor. As the first step, a joining node needs to contact some node already in the network to find its immediate successor. When a node receives the request, it will check whether the ID of the new node is between itself and its successor. If yes, the current successor is set to be the successor of the new node. Otherwise, the request is passed to a closest preceding finger.

Figure 4.1 shows our implementation of the join algorithm. In the implementation, we use a *State* record to keep information of predecessor, successor list and finger table. *successors* denotes a successor list and *fingers* is a finger table. Both are implemented with tuples (Erlang fixed-size arrays) of log N entries.

The function *is_element(X,A,B)* is used to check whether X is on an arc (A,B). It is equivalent to the mathematical operation $\in$ in the pseudo code of the algorithm. The function *closest_preceding_finger(Table,ID)* searches the local table for the highest predecessor of ID.

```
join(State,Gate) ->
    ID = State#peer.id,
    Gate ! {find_successor,ID},
    receive
        {find_successor,ID,Successor} ->
             New_State = State#peer{successors = setelement(1,State#peer.successors,Successor),
            fingers = setelement(1,State#peer.fingers,Successor)},
            loop(New_State)
    end.

loop
    {find_successor,Another_Peer} ->
            case is_element(Another_Peer,ID,Successor) of
                true ->
                        New_Successor = Another_Peer,
                        Another_Peer ! {find_successor,Another_Peer,Successor},
                        New_State = State#peer{
                                successors = setelement(1,State#peer.successors, New_Successor),
                                fingers = setelement(1,State#peer.fingers, New_Successor)},
                        loop(New_State);
                    false ->
                        Finger = closest_preceding_finger(State,Another_Peer),
                        Finger ! {find_successor,Another_Peer},
                        loop(State)
            end
```

Figure 4.1: Implementation of join algorithm

4.2. Data location

In a Chord system, data is stored on nodes and identified using unique numeric keys. To locate the node which is responsible for a key, we implement one operation: *lookup*. This operation takes a key as its argument and returns the identity of the node currently responsible for that key.

When a node receives a lookup request, it first checks whether the key is owned by itself, otherwise it will pass the request to the closest preceding finger. The process is repeated until the owner is found. The mechanism is similar to that of searching the successor of a joining node, except that the object of searching is a key of data, not an ID of node.

Figure 4.2 shows our implementation of the lookup algorithm.

```
{lookup,Key} ->
    case ID == Successor of
        case is_element(Key,ID,Successor) of
            true ->
                loop(State);
            false ->
                Finger = closest_preceding_finger(State,Key),
                Finger ! {lookup,Key},
                loop(State)
        end
    end;
```

Figure 4.2: Implementation of lookup algorithm

4.3. DHT logic

This module is used for a node to maintain DHT peers information and perform DHT logic. The DHT peers information includes predecessors, successor list and finger table. This module, thus, contains the implementation of stabilization, update successor list and update finger table algorithms.

In Erlang, they are implemented as periodic operations using a built-in function called `timer:apply_after` as shown in Figure 4.3a.

```
client(ID) ->
    start(ID),
    timer:apply_after(10,?MODULE,stabilize,[ID]),
    timer:apply_after(10,?MODULE,update_successors,[ID]),
    timer:apply_after(10,?MODULE,update_fingers,[ID]).
```

Figure 4.3a: Implementation of periodic operations

These three functions *stabilize*, *update_successors* and *update_fingers* are implemented as three external functions, which could be called to test separately.

The two functions *update_successors* and *update_fingers* are used to refresh the successor list and finger table. The function *stabilize* is used on concurrent node arrivals and departures.

Figure 4.3b shows in more details how the stabilization is implemented. When a node runs *stabilize*, it checks with its successor about the predecessor. The successor's predecessor is supposed to be the current node itself. If it is not the case, it means some node has just joined or left.

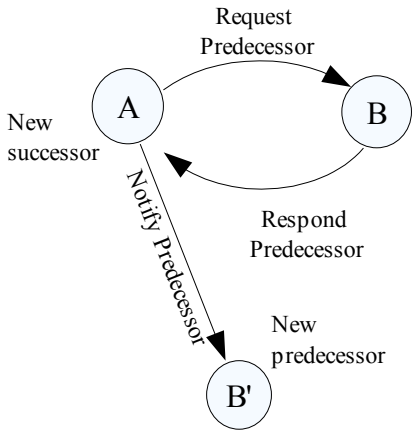


Figure 4.3b: Illustration of stabilization

Figure 4.3c shows our implementation of Chord stabilization algorithm, which includes request, respond and notify messages. A node changes its state when it detects a new successor or predecessor.

```
{request_predecessor,Another_Peer} ->
    Another_Peer ! {respond_predecessor, Predecessor},
    loop(State);

{respond_predecessor,Another_Peer} ->
    case is_element(Another_Peer,ID,Successor) of
        true ->
            New_Successor = Another_Peer,
            New_State = State#peer{
                    successors = setelement(1,State#peer.successors,New_Successor),
                    fingers = setelement(1,State#peer.fingers,New_Successor)},
            Another_Peer ! {notify_predecessor,ID},
            loop(New_State);
        false ->
            Successor ! {notify_predecessor,ID},
            loop(State)
    end;

{notify_predecessor,Another_Peer} ->
    case is_element(Another_Peer,Predecessor,ID) of
        true ->
            New_Predecessor = Another_Peer,
            case Successor == ID of
                true ->
                    New_Successor = Another_Peer,
                    Another_Peer ! {notify_predecessor,ID,Parent_ID},
                    New_State = State#peer{predecessor = New_Predecessor,
                        successors = setelement(1,State#peer.successors,New_Successor),
                        fingers = setelement(1,State#peer.fingers,New_Successor)},
                    loop(New_State);
                false ->
                    New_State = State#peer{predecessor = New_Predecessor},
                    loop(New_State)
            end;
        false ->
            loop(State)
    end;
```

Figure 4.3c: Implementation of stabilization algorithm

Chapter 5. **Testing with QuickCheck**

In this chapter, we present a case study of testing the Chord implementation presented in Chapter 4 against the model that we propose in Chapter 3, using the tool QuickCheck introduced in Chapter 2. In more details, we present how to use QuickCheck as a scheduler to simulate a Chord system and how to write properties, generate and execute test cases.

5.1. Simulation

Abstract model

We use QuickCheck to build the Chord model presented in Chapter 3. The model starts with an initial state of 1 - 9 inactive nodes and changes state according to six commands: *start*, *join*, *stop*, *stabilize*, *update_successors* and *update_fingers*. The feedbacks of these commands are then used to verify the implementation against the simulated model.

Figure 5.1a shows this setup in more details. When *stabilize(2)* is called, it initiates the external function *stabilize* of the implementation. We verify whether the feedback matches with the current state of the model, by checking the post condition of the command *stabilize*.

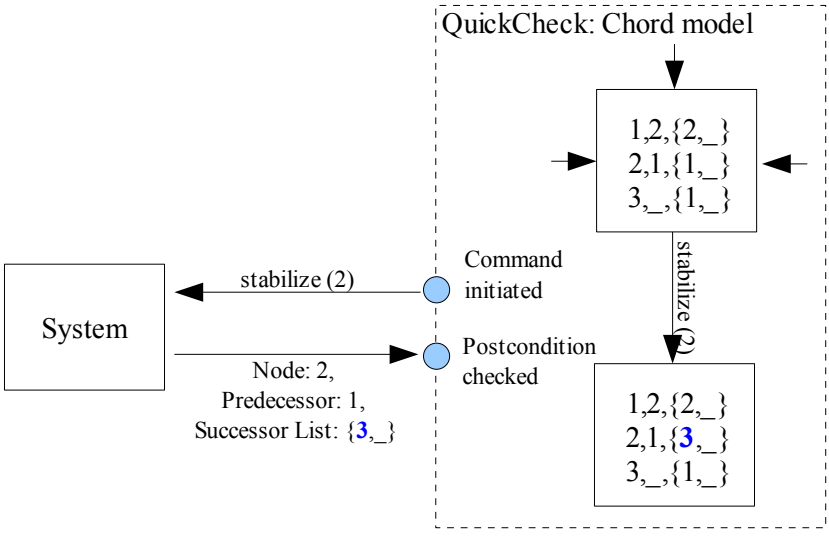


Figure 5.1a: Simulation of model

Note that all the relevant information in the model (nodes and relationships) are entirely calculated by the model itself. A model with three nodes 1, 2 and 4 active yields: 4 is the immediate successor of 2, 2 is the immediate successor of 1, 4 is the second successor of 1, and so on. Since those are the absolute positions, we can detect the cases of loopy networks (as warned in [24]).

Environment

DHTs function in a distributed and dynamic manner. Implementations of DHTs, therefore, have to be tested in a dynamic environment. It is supposed that these following situations could to be simulated:

(a) A node fails after its predecessor has just stopped;

(b) A node fails and then restarts (churn);

(c) A lookup occurs before stabilization; etc.

QuickCheck with its randomization support can simulate these situations by generating different sequences of events. Figure 5.1b shows the function to generate sequences of *start*, *join*, *stop*, *stabilize*, *update_successors* and *update_fingers*. The function *get_inactive* returns a list of inactive nodes, while *not_standalone* is true if there is more than one node currently active.

```
command(S) ->
  frequency(
     [{1,stop}] ++
     [{3,{call,?MODULE,start,[oneof(get_inactive(S#state.peers))]}}
          || all_inactive(S#state.peers)] ++
     [{5,{call,?MODULE,join,[oneof(get_inactive(S#state.peers)),oneof(get_active(S#state.peers))]}}
          || active_and_inactive(S#state.peers)] ++
     [{2,{call,?MODULE,stop,[oneof(get_active(S#state.peers))]}}
          || not all_inactive(S#state.peers)] ++
     [{10,{call,?MODULE,stabilize,[oneof(get_active(S#state.peers))]}}
          || not_standalone(S#state.peers)] ++
     [{10,{call,?MODULE,update_successors,[oneof(get_active(S#state.peers))]}}
          || not_standalone(S#state.peers)] ++
     [{10,{call,?MODULE,update_fingers,[oneof(get_active(S#state.peers))]}}
          || not_standalone(S#state.peers)] ).
```

Figure 5.1b: Simulation of environment

With this setting, the situation (a) could be simulated with two consecutive *stop* commands. The situation (b) could be simulated with a *stop* and a *start* of the same node in sequence. While the situation (c) can be: "..., *join(X)*, *lookup(X)*, *stabilize(X)*,...".

Assumptions

We use *precondition*'s to simulate the assumptions of Chord. An assumption like "a *stop* must not be called if it causes successor list of some node empty" could be simulated as follows (by checking that the node is not the last successor of *any* node).

```
precondition(S,{call,_,stop,[P,_]}) ->
    not empty_successor_list(S#state.peers,P).

empty_successor_list(List,P) -> ...
    lists:any(fun({_,_,Successors,_}) -> element(1,Successors) == P end, List).
```

Figure 5.1c: Simulation of assumptions

5.2. Property

We use QuickCheck to verify the correctness of user registration, data location and DHT logic. That is the property that we want the implementation to satisfy.

Nodes registered correctly

We verify that a command *join(N)* will lead to a correct registration of N. According to the model that we presented in Chapter 3, it means a new state with an additional node, and the immediate successor of that node must be correct. We verify that by checking the post-condition of that command.

Figure 5.2a shows how such a post condition is written in QuickCheck.

```
postcondition(S,{call,_,join,[P,_,_]},R) ->
    case R of
        {ok,P,_,Successor} ->
            Successor == successor(lists:sort(get_active(S#state.peers)++[P]),P);
        _ ->
            false
    end;
```

Figure 5.2a: Checking node registration

In this case, R is the response of *join*. This response message is sent from the running system to QuickCheck, with the format {ok,P,_,Successor}. That *Successor* of P will be checked with the *successor* of that P identified in the model side.

Data located correctly

We verify that a lookup correctly returns the node responsible for a given key. In Chord, key k is assigned to the first node whose identifier is equal to or follows k in the identifier space. Similar to the above case, this responsible node could be found from the model and checked with the response of the *lookup* command.

Chord logic functions correctly

We verify that *stabilize*, *update_successors* and *update_fingers* all perform correctly by checking their post-conditions.

Figure 5.2b shows how to verify the action *update_successors,* by checking the returning successor list with the modeled successors. A function *successors*, therefore, is needed to find among the currently active nodes, which are the entries of the successor list of P.

```
postcondition(S,{call,_,update_successors,[P,_]},R) ->
    R == successors(lists:sort(get_active(S#state.peers)),P);
```

Figure 5.2b: Checking Chord logic

5.3. Test cases

QuickCheck helps generate and execute random test cases of the correctness property. Each test case is a list of commands. In test generation, the commands are symbolic, which map symbolic variables to the results of symbolic calls. During test execution, these symbolic variables are replaced with their run-time values.

Figure 5.3 shows a sample test case:

```
[{init,{state,[{'1',undefined,
                    {undefined,undefined,undefined,undefined},
                    {undefined,undefined,undefined,undefined}},
               {'2',undefined,
                    {undefined,undefined,undefined,undefined},
                    {undefined,undefined,undefined,undefined}},
               {'3',undefined,
                    {undefined,undefined,undefined,undefined},
                    {undefined,undefined,undefined,undefined}}]}},
 {set,{var,1},{call,chord_eqc,start,[1]}},
 {set,{var,2},{call,chord_eqc,join,[2,1]}},
 {set,{var,3},{call,chord_eqc,stabilize,[2]}},
 {set,{var,4},{call,chord_eqc,join,[3,2]}},
 {set,{var,5},{call,chord_eqc,update_successors,[3]}},
 {set,{var,6},{call,chord_eqc,stabilize,[3]}},
 {set,{var,7},{call,chord_eqc,lookup,[3,4]}},
 {set,{var,8},{call,chord_eqc,lookup,[3,4]}},
 {set,{var,9},{call,chord_eqc,stop,[1]}},
 {set,{var,10},{call,chord_eqc,update_successors,[3]}},
 {set,{var,11},{call,chord_eqc,update_successors,[2]}},
 {set,{var,12},{call,chord_eqc,update_fingers,[3]}},
 {set,{var,13},{call,chord_eqc,stabilize,[2]}},
 {set,{var,14},{call,chord_eqc,stabilize,[2]}},
```

Figure 5.3: A sample test case

This test case starts with three inactive nodes 1, 2 and 3. Their predecessors, successors and fingers are all undefined initially. The first command starts node 1, and set the result to var 1. The second command is then called to join node 2 via node 1 and set var 2 to the result. The third command is to stabilize node 2 and so on. A sample test case like this could be printed out to check whether the preconditions and other assumptions have been simulated correctly.

In the next chapter, we present the result when we apply such test cases to test our Chord implementation.

Chapter 6. **Results and analysis**

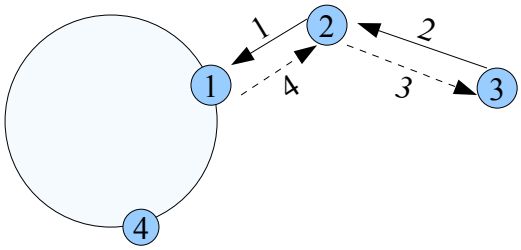
In this chapter, we present and analyze the result of our testing. We show several serious bugs in the current implementation. We also show in which situations (test cases), these bugs are detected. Based on the analysis, we then suggest some modifications in the original Chord algorithm. As a result, we introduce a modified version of it.

6.1. Faults found

The first implementation contains several bugs, four of which are quite serious.

Join via another joining node

This is a fault when a node tries to join the existing network via another joining node. Figure 6.1a shows a scenario when node 3 tries to join the network via node 2, which by that time is still waiting for feedback. This scenario leads to a creation of one more network with two nodes 2 and 3.



1- Node 2 is requesting its immediate successor
2 - Node 3 requests its immediate successor via node 2
3 - At that time, node 2 has no successor yet, so it considers node 3 as its successor. Node 3, in turn, considers node 2 as its successor. Nodes 2 and 3 now forms a separated network of their own.
4 - The existing network finds out the immediate successor of 2. But it is too late now.

Figure 6.1a: Join via another joining node

The fault is serious but quite difficult to find out, since every request, in this case, has a reply as supposed (4 for 1 and 3 for 2). The separation is also hard to observe if we consider one node in a relation with just one or two other nodes relatively.

In this case, our system-level model has shown its advantages. We detect the separation by checking the response from 2 to 3. The response says 2 is the successor of 3, while according to our model, the successor of node 3 in a network of four active nodes 1, 2, 4 and 3 must be node 4 instead. We detect the error by checking the post condition of *join(3,2)*.

Figure 6.1b shows the QuickCheck printout of the fault. In this case, we simulate a network of 200 nodes. The problem happens when node 120 joins via node 98, which itself is currently joining.

```
 Res: {postcondition,false}
Shrinking.....(15 times)
200
0
1
[{init,{state,[{1,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}},

              {200,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}}],
             '1',
             0}},
 {set,{var,1},{call,chord_eqc,start,[127]}},
 {set,{var,5},{call,chord_eqc,join,[98,127]}},
 {set,{var,8},{call,chord_eqc,join,[120,98]}}]
```

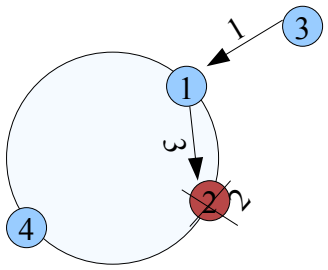
Figure 6.1b: Printout of fault: join via another joining node

To fix this error, we distinguish reply messages from request ones. It means instead of using {find_successor,_} as in the algorithm presented by Bakhshi and Gurov and in our Chord implementation, we use {request_successor,_} and {respond_successor,_}. By doing that, when node 2 (in Figure 6.1a) receives messages from 1 and 4, it can choose to ignore those from the latter.

Joining when gate or immediate node fails

This fault was detected after we had already fixed the first one. The problem is when the gate of a joining node or some of its fingers fails unexpectedly. It leads to an infinite waiting time for the joining node. The fault is detected when we set a time-out for joining, and receive a false message instead of a correct immediate successor as modeled

Figure 6.1c shows the scenario in more details. Figure 6.1d shows the QuickCheck printout of the fault.



1- Node 3 requests its immediate successor.

2 - Node 2 suddenly fails

3 - Node 1 still considers node 2 as the closest preceding finger, so it forwards the request from node 3 to a non-existing node 2. Node 3 will never receive any reply to be able to join the existing network

Figure 6.1c: Joining when gate or its closest preceding finger fails

```
 {set,{var,37},{call,chord_eqc,update_successors,[104]}},
 {set,{var,38},{call,chord_eqc,stop,[139]}},
 {set,{var,39},{call,chord_eqc,update_successors,[145]}},
 {set,{var,40},{call,chord_eqc,stop,[10]}},
 {set,{var,41},{call,chord_eqc,update_successors,[145]}},
 {set,{var,42},{call,chord_eqc,join,[57,145]}},
 {set,{var,43},{call,chord_eqc,stabilize,[57]}},
 {set,{var,44},{call,chord_eqc,update_successors,[145]}},
 {set,{var,45},{call,chord_eqc,stabilize,[104]}},
...
 Res: {postcondition,false}
Shrinking.....(15 times)
200
0
1
[{init,{state,[{1,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}},
              {200,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}}],
             '1',
             0}},
 {set,{var,1},{call,chord_eqc,start,[71]}},
 {set,{var,9},{call,chord_eqc,join,[10,71]}},
 {set,{var,19},{call,chord_eqc,join,[145,10]}},
 {set,{var,42},{call,chord_eqc,join,[57,145]}}]
```

Figure 6.1d: Printout of fault: Joining when gate or immediate node fails

In this scenario, the post condition of "join 57 via 145" is false. The reason is because node 10 as the closest preceding finger of 57 has stopped before that (at var 40).

The problem is quite easy to trace out but complicated to fix. The root of this problem lies in the way we choose periodic recovery over reactive recovery. In DHT literature, e.g. [21], it is recommended to choose the former method, since it is considered to be more efficient under high churn rates. In Erlang environment, however, we tend to use the latter one thanks for the built-in reactive detection mechanisms (process linking and monitoring). Back to the problem that we are facing here, we realize that the use of reactive recovery can fix the problem, while the periodic recovery method cannot (the gate has to wait until the next stabilization to realize the failure). We, therefore, tentatively propose a solution to this problem by creating monitored links from a node to all its fingers and successors, so that the monitored node can actively and quickly be informed about the failures and take actions accordingly.

By doing so, after node 2 fails (in Figure 6.1c), node 1 will be informed and quickly choose node 4 as a replacement to forward the request to. Node 3 can then join the network with node 4 as its immediate successor.

All successors and fingers simultaneously fail

The problem is because each node stores just a limited number of other nodes as its successors in the successor list and the same limited number of nodes as fingers in the finger table. There is a chance that all of them simultaneously fail, that leads the node to be isolated completely from the existing network. Although the chance for this to happen is quite small, it is worth noticing. We show that our implementation also experiences this problem. We highlight that a simulated network with 2 - 8 nodes has a better chance to come across this problem since the size of their successor lists and finger tables are normally 2 or 3.

Figure 6.1e shows such a scenario in a network of 4 active nodes

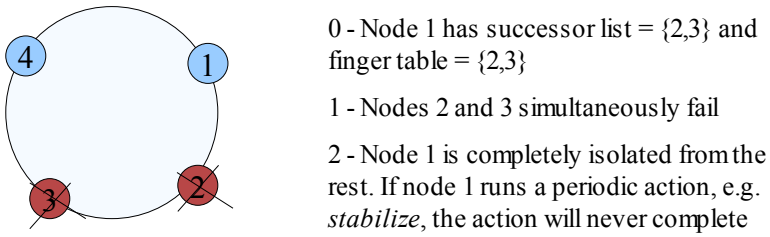


Figure 6.1e: All successors and fingers simultaneously fail

The fault is detected when we set time-out for the maintenance commands, and a false message is received instead of an informative message as expected.

We show how QuickCheck can detect this error in Figure 6.1f. It is a simulated network with three nodes, two of which (nodes 1 and 2) are currently active and running. Suddenly, node 2 stops. That causes the stabilization at node 1 fails.

```
[{init,{state,[{1,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}},
              {2,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}},
              {3,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}}],
             '1',
             0}},
 {set,{var,1},{call,chord_eqc,start,[1]}},
 {set,{var,2},{call,chord_eqc,stop,[1]}},
 {set,{var,3},{call,chord_eqc,start,[2]}},
 {set,{var,7},{call,chord_eqc,join,[1,2]}},
 {set,{var,16},{call,chord_eqc,stop,[2]}},
 {set,{var,17},{call,chord_eqc,stabilize,[1]}}]
State: {state,[{1,
               undefined,
               {2,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}},
              {2,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}},
              {3,
               undefined,
               {undefined,undefined,undefined,undefined},
               {undefined,undefined,undefined,undefined}}],
             '1',
             0}
 Res: {postcondition,false}
false
```

Figure 6.1f: Printout of fault: all successors and fingers simultaneously fail

This fault cannot be fixed at the implementation level, since by design every DHT just maintains a small list of neighbors. One thing that we could do, however, is to consider that as a specification assumption. In other words, we set a precondition so that *stop* must not be called if it causes the successor list of some node empty. Figure 5.1c has shown such a precondition.

Furthermore, as before, we tentatively propose a solution to this problem by using the reactive recovery mechanism. Using that, the system can quickly detect and recover failures one by one, instead of waiting until the next stabilization to realize that all the successors/ fingers have been failed (is that what we call “simultaneous”?).

Old messages come back

The problem happens when a node sends out a request before leaving, and when it joins back, the response message returns. And the information delivered by the response message, by that time, does not match with the current situation of the network any more. That causes a wrong update for the leave-and-rejoin node, leading it to function unexpectedly. In our testing, we observed the case when a node stops and then re-joins, its immediate successor is correct but one or two entries in the successor list are wrong, implying that the response of a previous *update_successors* command has updated the list with old entries.

Figure 6.1g shows a failing test case when node 4 stops, restarts and runs *update_successors*. The post condition of the *update_successors* is false.

```
 Res: {postcondition,false}
false
...
             {10,
              undefined,
              {undefined,undefined,undefined,undefined},
              {undefined,undefined,undefined,undefined}}],
            '1',
            0}},
 {set,{var,7},{call,chord_eqc,start,[4]}},
 {set,{var,19},{call,chord_eqc,join,[8,4]}},
 {set,{var,30},{call,chord_eqc,stop,[4]}},
 {set,{var,35},{call,chord_eqc,join,[4,8]}},
 {set,{var,37},{call,chord_eqc,update_successors,[4]}}]
```

Figure 6.1g: Printout of fault: old messages come back

6.2. Modified algorithm

We have shown four serious bugs in our first implementation. Different from other less serious ones, which we have already fixed in the code at the implementation level, these four bugs require changes in the original algorithm.

The changes that we introduce in Figure 6.2 are in comparison with the Chord algorithm proposed by Stoica et al. [1]; especially the changes in join and stabilization are in comparison with those presented by Bakhshi and Gurov [13].

Basically, we have made the following modifications:

- Separating request from respond messages and including the sender to all the request ones
- Creating monitored links between node and its predecessor (replacing *check_predecessor()* in [1])
- Creating monitored links between node and its immediate successor
- Adding *update_successors* as another stabilization function (not formally defined by [1])
- Modifying join: a new node is informed of not only its immediate successor, but also its predecessor and the successor list of its new immediate successor

All these modifications are aimed to

- Make the communication between a source and a destination clean and unambiguous
- Quickly detect and recover failures of closely neighboring nodes
- “Divide and conquer” simultaneous failures
- With the same number of messages, update a new node as much information as possible and inform its existence to as many other nodes as possible

Our modified Chord algorithm is presented in the syntax of the Erlang programming language. In Figure 6.2, *Size* is used to denote the size of the successor list and the finger table. It corresponds to an identifier space of 2^{Size}. *Starts* implies a list of numbers calculating as node id $+ 2^{k-1}$, where k is in between 1 and *Size*. *Start* is also used to denote the index of an entry in a tuple. X $\in$ (A,B) denotes that X is on an arc (A,B). We use “+” to concatenate two tuples, and “-” to subtract one entry from a tuple.

```
start(N) ->
   Predecessor = N,
   element(1,Successors) = N,
   element(1,Fingers) = N,
   process_flag(trap_exit,true),
   loop(N,Predecessor,Successors,Fingers);

join(N,Gate) ->
   Gate ! {request_neighbors,N},
   receive
      {respond_neighbors,N,P,Ss} ->
          Predecessor = P;
          Successors = Ss,
          Successor = element(1,Successors),
          monitor(process,Predecessor),
          monitor(process,Successor),
          process_flag(trap_exit,true),
          loop(N,Predecessor,Successors,Fingers)
   end.

closest_preceding_finger(Fingers,Id,N) ->
   Finger = element(N,Fingers),
   case finger ∈ (N,Id] of
      true ->
          Finger;
      false ->
          closest_preceding_finger(Fingers,Id,N-1)
   end.

loop(N,Predecessor,Successors,Fingers) ->
   Successor = element(1,Successors),
   receive

      {request_neighbors,N'} ->
          case N' ∈ (N,Successor] ->
             true ->
                N' ! {respond_neighbors,N',N,Successors);
             false ->
                Finger = closest_preceding_finger
                                          (Fingers,N',Size),
                Finger ! {request_neighbors,N'}
          end,
          loop(N,Predecessor,Successors,Fingers);

      {request_predecessor,N'} ->
          N' ! {respond_predecessor,Predecessor},
          loop(N,Predecessor,Successors,Fingers);

      {respond_predecessor,N'} ->
          case N' ∈ (N,Successor) of
             true ->
                New_Successor = N',
                monitor(process,New_Successor),
                New_Successor ! {notify_predecessor,N},
                New_Successors =
                  setelement(1,Successors,New_Successor),
                loop(N,Predecessor,New_Successors,Fingers);
             false ->
                Successor ! {notify_predecessor,N},
                loop(N,Predecessor,Successors,Fingers)
          end;

      {notify_predecessor,N'} ->
          case Predecessor == N or N' ∈(Predecessor,N) of
             true ->
                New_Predecessor = N',
                monitor(process,New_Predecessor),
                loop(N,New_Predecessor,Successors,Fingers);
             false ->
                loop(N,Predecessor,Successors,Fingers)
          end;

      {request_successors,N'} ->
          N' ! {respond_successors,N,Successors},
          loop(N,Predecessor,Successors,Fingers);

      {respond_successors,N,Next_Successors} ->
          New_Successors = Successor +
             Next_Successors - element(last,Next_Successor),
          loop(Predecessor,New_Successors,Fingers);

      {request_finger,N',Start} ->
          case Start ∈ (Id,Successor) of
             true ->
                N' ! {respond_finger,Start,Successor};
             false ->
                Finger = closest_preceding_finger
                                          (Fingers,Start,Size),
                Finger ! {request_finger,N',Start}
          end,
          loop(N,Predecessor,Successors,Fingers);

      {respond_finger,Start,Successor} ->
          New_Fingers = setelement(Start,Fingers,Successor),
          loop(N,Predecessor,Successors,New_Fingers);

      {'DOWN',_,Successor,_} ->
          New_Successor = element(2,Successors),
          monitor(process,New_Successor),
          loop(N,Predecessor,Successors,Fingers);

      {'DOWN',_,Predecessor,_} ->
          New_Predecessor = N,
          loop(N,Predecessor,Successors,Fingers);

      after Time ->
          Successor ! {request_predecessor,N},
          Successor ! {request_successors,N},
          [closest_preceding_finger(Fingers,Start,Size) !
                        {request_finger,N} || Start <- Starts],
          loop(N,Predecessor,Successors,Fingers)

end.
```

Figure 6.2: Modified Chord algorithm

Chapter 7. **Discussion**

In this chapter, we analyze the result in a broader context. We discuss about the applicability of the proposed testing method to testing other DHTs, such as CAN and Tapestry. We then compare it with the tracing method. A little graph-theoretic analysis of our model is next presented.

7.1. CAN, Tapestry and other DHTs

The finite state machine that we propose to model the Chord protocol could be used to model other DHTs. By decomposing a DHT into user registration and DHT logic modules, we could model each of them in the same way that we model the Chord protocol.

Testing the user registration is to ensure that all the arrivals and departures perform correctly. A registration is called a success if the user is inserted in the correct position. In Chord, it means correct immediate successor. In CAN, it implies correct zone. While in Tapestry, it includes correct branch and leaves.

Figure 7.1a shows a model of CAN registration.

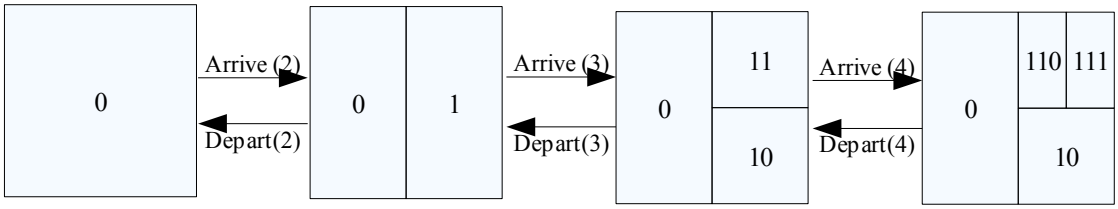


Figure 7.1a: Model of 2-d CAN registration

The state, in this case, is defined as a collection of many individual nodes, each of which stores chunk (zone) of the entire space. When a new node arrives, an existing zone is split into two halves, one of which is assigned to the new node. The split is done by following an order of dimensions, e.g. horizontal – vertical – horizontal and so on. Using our testing method, one can write a *postcondition* to check that zones are split correctly when a new node arrives. Similarly, we can check that the zones are correctly merged when a node departs. This finite state machine could be extended to cover the CAN recovery algorithm.

Similarly, Tapestry registration could be modeled with a finite state machine as shown in Figure 7.1b.

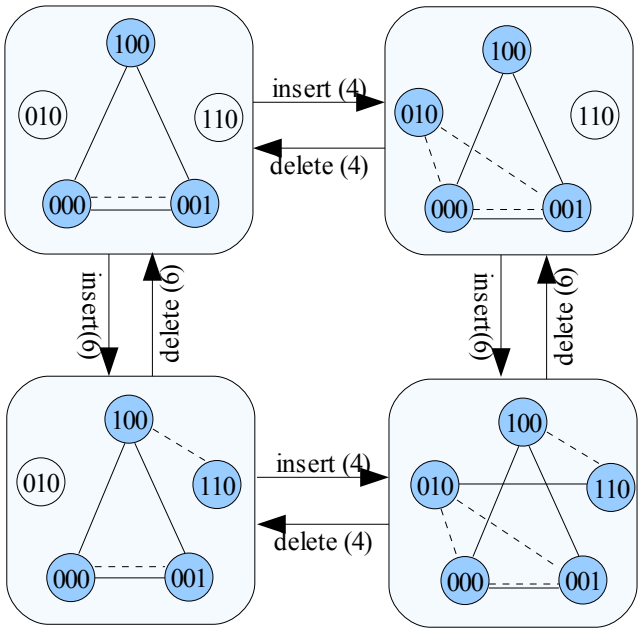


Figure 7.1b: Model of Tapestry registration

In this model, a continuous line denotes a relationship between a node and one of its level 1 neighbors; and a dotted line denotes a relationship between a node and one of its level 2 neighbors. A node is considered as a level i neighbor of another one if their i^{th} digits match.

By using this finite state machine, we can check whether an *insert* command leads to correct relationships between a new node and its level 1 and 2 neighbors. Or we can check the post condition of the *delete* command to verify that the total number of active nodes is reduced by one. A finite state machine like this could be extended to include the internal logic of Tapestry, helping troubleshoot sophisticated problems.

7.2. Tracing

The finite state machine that we have proposed is in the top system level. Our testing method, therefore, is based on the observation about the changes in nodes and their relationships.

There is another way that we can test the system by observing the sequences of communication messages. With this approach, we trace all the messages sent and received when the system is running. We also trace all the events when nodes change their own states (e.g. starting, joining, joined, etc.). All the trace data will then be analyzed to check whether these messages and events are sent, received and happened in a correct (valid) sequence.

Figure 7.2 shows a valid sequence of messages and events of joining.

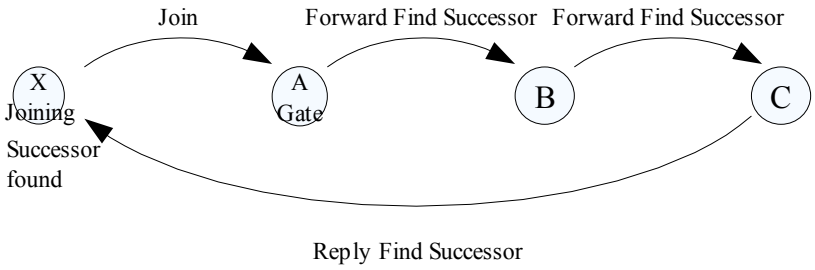


Figure 7.2: Messages and events of joining

When we receive a message “Reply Find Successor from C to X”, for example, we check in the trace record, whether there has already been a message “Forward Find Successor from some node to C”. If not, it means there was some error in this joining activity.

This testing technique seems to be efficient to test continually running and asynchronous message passing systems, since it lets the system run freely and analyses the events in a backward direction.

The disadvantage of this approach, however, is that it views a message and an event in a relation with another one, not with the whole network. So, as shown in the previous chapter, when a ring is separated into two and the sequence of messages is still valid, this testing method does not consider that as a fault. This “relative” problem also occurs if we test a peer-to-peer system based on a single-peer model.

7.3. Graph theory for structured peer-to-peer systems

The underlying philosophy of our model is to view an infinite state machine as a collection of finite state machines, and view a multi-dimensional model as several two-dimensional ones. This philosophy still applies when we study graphs of structured peer-to-peer systems.

We understand that any DHT system could be drawn as a k-node-connected graph. We suggest drawing it as k one-node-connected graphs. A graph theory for structured peer-to-peer systems, therefore, could be a theory of *combined* graphs. In the algebra terminology, we need addition (or multiplication) operations for graphs.

Some initial questions could be

(1) What is the average path length of a graph having by combining a star-like graph and a ring-like graph? (knowing that a star has the average path length of 1 and that of a N-node ring is N/2)

(2) How is a graph modeled as (a) set(s)?

(3) Which graph shape provides a uniform distribution of congestion/ node degree?

Such studies would be the new directions for future research.

Figure 7.3 shows how a mesh-like graph of Chord is drawn as four one-node-connected ones.

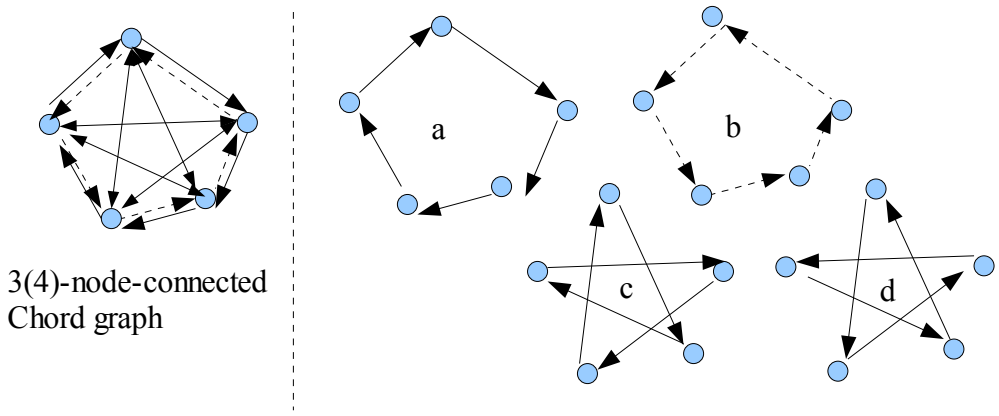


Figure 7.3: Chord graphs: (a) node and the 1st successor, (b) node and predecessor, (c) node and the 2nd successor, (d) node and the 3rd successor

Chapter 8. **Conclusion**

This thesis has introduced a novel method for testing implementations of DHTs. As a contribution, we have proposed abstract state machines as the ways to model DHT-based peer-to-peer systems, and use them as the correctness properties to test implementations of DHTs. We have also proposed to separate user registration and location services from DHT logic, and have shown that the separation make it easier to troubleshoot problems.

As a case study, we have tested an implementation of Chord protocol, one of the well-known DHTs, using QuickCheck, a property-based tool for random testing. Using this testing method, we have detected several serious faults in the current implementation. We, furthermore, have demonstrated how the faults have been found and troubleshot.

We suggested some changes in the original Chord algorithm. Significantly, we have introduced a modified version of Chord algorithm, with a combination of reactive and periodic recovery mechanisms. We argue that the use of reactive recovery could reduce the faults of simultaneous failures. A new version of Chord implementation based on the modified algorithm has been produced and included with this document as an appendix.

As future works, we would like to see the application of this method to testing implementations of other DHTs. A study about its efficiency in comparison with other methods is also a need. Specially, we are looking forward to a graph-theoretic analysis of the model in particular and of structured peer-to-peer systems in general.

References

Normative references

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pp 149-160, ACM Press, 2001.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Schenker, A scalable content addressable network. *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, ACM Press, 2001, pp. 161-172.

[3] A. Rowstron and P. Druschel, Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218: 329-250, 2001.

[4] B. Zhao, J. Kubiatowicz and A. Joseph, Tapestry: An infrastructure for fault-tolerant wide-area location and routing, UC Berkeley, Tech. Rep. UCB/CSD-01-1141, April 2001.

[5] D. Malkhi, M. Naor, and D. Ratajczak, Viceroy: A Scalable Dynamic Emulation of the Butterfly. *21st ACM Symposium on Principles of Distributed Computing*, California, 2002.

[6] P. Maymounkov and D. Mazieres, Kademlia: A peer-to-peer Information Systems based on the XOR Metric. *1st International workshop on Peer-to-Peer Systems (IPTPS '02), 2002.*

[7] T. Arts and J. Hughes, QuickCheck for Erlang. *Proceedings of the 9th Erlang User Conference*, Stockholm, 2003.

[8] T. Arts, K. Claessen and H. Svensson, Semi-formal development of a fault-tolerant leader election protocol in Erlang, *Lecture Notes in Computer Science*, 3395:140-154, January 2005.

[9] T. Arts and H. Svensson, A new leader election implementation, *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, ACM Press 2005.

[10] T. Arts, K. Claessen, J. Hughes and H. Svensson, Testing Implementations of Formally Verified Algorithms, In *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*, 2005.

[11] T. Arts, J. Hughes, U. Wiger and J. Johansson, Testing telecoms software with Quviq QuickCheck. *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*.

[12] K. Claessen and J. Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs, *ICFP*, pp 268-279, 2000.

[13] R. Bakhshi and D. Gurov, Verification of Peer-to-Peer Algorithms: A Case Study, School of Information and Communication Technology, Royal Institute of Technology, Kista, Sweden

[14] J. Armstrong, R. Virding, C. Wikstrom and M. Williams, *Concurrent Programming in Erlang,* Prentice Hall, 2nd edition, 1996

[15] F. Araujo and L. Rodrigues, Survey on DHTs, University of Lisbon, 2006.

Informative references

[16] J. Armstrong, Making reliable distributed systems in the presence of software errors, PhD Thesis. Swedish Institute of Computer Science, 2003.

[17] K. Singh and H. Schulzrinne, Peer-to-peer Internet Telephony using SIP, *Proceedings of the 2005 Network and Operating Systems Support for Digital Audio and Video Workshop (NOSSDAV) '05* , June 2005

[18] G. Tretmans and A. Belinfante, Automatic Testing with Formal Methods. Technical Report TR-CTIT-99-17 Center for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625.

[19] G. Fink and M. Bishop, Property-Based Testing: A New Approach to Testing for Assurance, *ACM SIGSOFT Software Engineering Notes* 22(4) pp. 74–80, July 1997.

[20] J. Li, J. Stribling, T. Gil, R. Morris and F. Kaashoek, Comparing the performance of distributed hash tables under churn, *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, February 2004.

[21] S. Rhea, D. Geels, T. Roscoe and J. Kubiatowicz, Handling churn in a DHT, University of California at Berkeley, Tech. Rep., December 2003.

[22] C.G. Plaxton, R. Rajaraman and A.W. Richa, Accessing nearby copies of replicated objects in a distributed environment*, ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 311-320

[23] L. Garces-Erice, K. Ross, E. Biersack, R. Felber and G. Urvoy-Keller, Topology-centric lookup service, *COST264/ACM Fifth International Workshop on Networked Group Communications (NGC)*, Munich, Germany, 2003

[24] D. Liben-Nowell, H. Balakrishnan and D. Karger, Analysis of the evolution of peer-to-peer systems, *PODC: Proceedings of the 21st annual symposium on Principles of distributed computing*. NY, USA: ACM Press 2002, pp 233-242.

[25] P. Duchon, N. Hanusse, E. Lebhar and N. Schabanel, Could any graph be turned into a small-world?, *Special issue of the international journal Theoretical Computer Science on Complex Networks*, 2005.

[26] S. Saroiu, P. Gummadi and S. Gibble, A measurement study of peer-to-peer file sharing systems, Scalability and Traffic Control in IP Networks II, ser. *Proceedings of SPIE*, vol. 4868, July 2002

[27] I. Gupta, K. Birman, P. Linga, A. Demers and R. van Renesse, Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead, *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*.

[28] C. Edwards, A. Harwood and E. Tanin, Network virtualisation for transparent testing and experimentation of distributed applications*, IEEE International Conference on Networks*, volume 2, pages 1089-1094, November 2005.
[29] L. Navslavsky, D. Richardson and H. Ziv, Scenario based and State Machine based Testing: An Evaluation of Automated Approaches, ISR Technical Report #UCI-ISR-06-13, University of California

[30] D. Loguinov, J. Casas and X. Wang, Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience, *IEEE/ACM Transactions on Networking*, pp 1107-1120, 2005.

[31] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker and I. Stoica, The impact of DHT routing geometry on resilience and proximity, in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, USA, ACM Press, 2003, pp.381 -394.

[32] U. Wieder, U. and M. Naor, A simple fault tolerance distributed hash table, in *Proceedings of IPTPS*, February 2003.

[33] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, R. Panighahy, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654-663.

[34] Erlang http://www.erlang.org

[35] Eclipse http://www.eclipse.org

[36] Erlide: Eclipse plug in for Erlang http://erlide.sourceforge.net

[37] A P2P SIP implementation http://p2psipphone.sourceforge.net

Appendices

Appendix A: Chord specification

```
%----------------------------------------------------------------------------------------------------
% File              : chord_eqc.erl
% Description       : A QuickCheck specification of Chord
% Last modified : 30 March, 2007
%----------------------------------------------------------------------------------------------------

-module(chord_eqc).

%----------------------------------------------------------------------------------------------------
% Include files
%----------------------------------------------------------------------------------------------------

-include("eqc.hrl").
-include("eqc_statem.hrl").

%----------------------------------------------------------------------------------------------------
% Definitions
%----------------------------------------------------------------------------------------------------

-define(space_size,4).

%----------------------------------------------------------------------------------------------------
% Records
%----------------------------------------------------------------------------------------------------

-record(state,{peers = [],key,time}).

%----------------------------------------------------------------------------------------------------
% External exports
%----------------------------------------------------------------------------------------------------

-compile(export_all).

-export([command/1, next_state/3, precondition/2, postcondition/3,
         prop_chord/0, initial_state/0,initial_state/3,
         successor/2,predecessor/2,successors/2,finger/3,fingers/2
    ]).

%----------------------------------------------------------------------------------------------------
% Behaviour
%----------------------------------------------------------------------------------------------------

-behavior(eqc_statem).

%====================================================================
% QuickCheck functions
%====================================================================

%----------------------------------------------------------------------------------------------------
% Function: prop_chord/1
% Descrip.: Checking the correctness of registration service, lookup service and Chord logic
% Returns : bool()
%----------------------------------------------------------------------------------------------------

prop_chord() ->
```

```
    ?FORALL(NrPeers,choose(1,9),
      ?FORALL(Time,choose(1,100),
        ?FORALL(Key,choose(1,9),
          ?FORALL(Cmds,commands(?MODULE,initial_state(NrPeers,Key,Time)),
            begin
              {_,S,R} = run_commands(?MODULE,Cmds),
              [stop(P,0) || P <- get_active(S#state.peers), not all_inactive(S#state.peers)],
              ?WHENFAIL(io:format("State: ~p\n Res: ~p\n",[S,R]),
                R == ok
              )
            end)))).

initial_state() ->
    #state{peers = []}.

initial_state(N,Key,Time) ->
    lists:foldl(fun(X,S) ->
        ID = list_to_atom(integer_to_list(X)),
        Predecessor = undefined,
        Successors = erlang:make_tuple(?space_size,undefined),
        Fingers = erlang:make_tuple(?space_size,undefined),
        S#state{peers = S#state.peers ++ [{ID,Predecessor,Successors,Fingers}],
                key = list_to_atom(integer_to_list(Key)),  time = Time}
    end,#state{},lists:seq(1,N)).

command(S) ->
  frequency(
        [{1,stop}] ++
        [{3,{call,?MODULE,start,[oneof(get_inactive(S#state.peers)),S#state.time]}}
                        || all_inactive(S#state.peers)] ++
        [{5,{call,?MODULE,join,[oneof(get_inactive(S#state.peers)),oneof(get_active(S#state.peers)),
                        S#state.time]}} || active_and_inactive(S#state.peers)] ++
        [{2,{call,?MODULE,stop,[oneof(get_open(S#state.peers)),S#state.time]}}
                        || not all_inactive(S#state.peers)] ++
        [{20,{call,?MODULE,stabilize,[oneof(get_active(S#state.peers)),S#state.time]}}
                        || not_standalone(S#state.peers)] ++
        [{20,{call,?MODULE,update_successors,[oneof(get_active(S#state.peers)),S#state.time]}}
                        || not_standalone(S#state.peers)] ++
        [{5,{call,?MODULE,update_fingers,[oneof(get_active(S#state.peers)),S#state.time]}}
                        || not_standalone(S#state.peers)] ++
        [{10,{call,?MODULE,lookup,[oneof(get_active(S#state.peers)),S#state.key,S#state.time]}}
                        || not_standalone(S#state.peers)]
        ).

%---------------------------------------------------------------------------------------------------------
% Command       : start/2
% Description   : Checking a node starts successfully
% Precondition  : The node must be inactive
% Postcondition : The function must return true
%---------------------------------------------------------------------------------------------------------

start(Peer,Time) ->
  wait(Time),
  chord:start(Peer).

%---------------------------------------------------------------------------------------------------------
% Command       : join/3
% Description   : Checking a node joins successfully
% Precondition  : The node must be inactive, and the node must be active
% Postcondition : The node must be registered in a correct position: correct predecessor and successor
%---------------------------------------------------------------------------------------------------------
```

```
join(Peer,Gate,Time) ->
   wait(Time),
   chord:start(Peer,Gate).

%----------------------------------------------------------------------------------------------------------
% Command       : stop/2
% Description   : Checking a node stops successfully
% Precondition  : The node must be active
% Postcondition  : The function must return true
%----------------------------------------------------------------------------------------------------------

stop(Peer,Time) ->
   wait(Time),
   chord:stop(Peer).

%----------------------------------------------------------------------------------------------------------
% Command       : stabilize/2
% Description   : Checking a node and its neighbors are stablized successfully
% Precondition  : The node must be active
% Postcondition  : The predecessor of its successor must be the node itself
%----------------------------------------------------------------------------------------------------------

stabilize(Peer,Time) ->
   wait(Time),
   chord:stabilize(Peer).

%----------------------------------------------------------------------------------------------------------
% Command       : update_successors/2
% Description   : Checking a node refreshes its successor list successfully
% Precondition  : The node must be active
% Postcondition  : The first entries in the successor list must be correct
%----------------------------------------------------------------------------------------------------------

update_successors(Peer,Time) ->
   wait(Time),
   chord:update_successors(Peer).

%----------------------------------------------------------------------------------------------------------
% Command       : lookup/3
% Description   : Checking the lookup service
% Precondition  : The node must be active
% Postcondition  : Keys are found in the correct node
%----------------------------------------------------------------------------------------------------------

update_fingers(Peer,Time) ->
   wait(Time),
   chord:update_fingers(Peer).

lookup(Peer,Key,Time) ->
   wait(Time),
   chord:lookup(Peer,Key).

precondition(S,{call,_,start,_}) ->
    all_inactive(S#state.peers);

precondition(S,{call,_,join,[P,G,_]}) ->
    lists:member(P,get_inactive(S#state.peers)) and lists:member(G,get_active(S#state.peers));

precondition(S,{call,_,stop,[P,_]}) ->
    lists:member(P,get_active(S#state.peers));

precondition(S,{call,_,stabilize,[P,_]}) ->
```

```
    lists:member(P,get_active(S#state.peers));

precondition(S,{call,_,update_successors,[P,_]}) ->
    lists:member(P,get_active(S#state.peers));

precondition(S,{call,_,update_fingers,[P,_]}) ->
    lists:member(P,get_active(S#state.peers));

precondition(S,{call,_,lookup,[P,_,_]}) ->
    lists:member(P,get_active(S#state.peers)).

postcondition(_,{call,_,start,_},R) ->
    R == true;

postcondition(S,{call,_,join,[P,_,_]},R) ->
    case R of
        {ok,P,Predecessor,Successor} ->
            Predecessor == predecessor(lists:sort(get_active(S#state.peers)++[P]),P) and
                (Successor == successor(lists:sort(get_active(S#state.peers)++[P]),P));
        _ ->
            false
    end;

postcondition(_,{call,_,stop,_},R) ->
    R == true;

postcondition(S,{call,_,stabilize,[_,_]},R) ->
    case R of
        {ID,Predecessor} ->
            Predecessor == predecessor(lists:sort(get_active(S#state.peers)),ID);
        _ ->
            false
    end;

postcondition(S,{call,_,update_successors,[P,_]},R) ->
    element(1,R) == element(1,successors(lists:sort(get_active(S#state.peers)),P));

postcondition(S,{call,_,update_fingers,_},R) ->
    case R of
        {_,Start,Finger} ->
            Finger == finger(get_active(S#state.peers),Start);
        false ->
            true;
        _ ->
            false
    end;

postcondition(S,{call,_,lookup,[_,K,_]},R) ->
    case R of
        {K,Peer} ->
            Peer == finger(get_active(S#state.peers),K);
        _ ->
            false
    end.

next_state(S,_,{call,_,start,[Id,_]}) ->
    {value,{Id,Predecessor,Successors,Fingers}} = lists:keysearch(Id,1,S#state.peers),
    NewPeers = lists:keyreplace(Id,1,S#state.peers,{Id,Predecessor,setelement(1,Successors,Id),Fingers}),
    S#state{peers = NewPeers};

next_state(S,_,{call,_,join,[Id,_,_]}) ->
    {value,{Id,Predecessor,Successors,Fingers}} = lists:keysearch(Id,1,S#state.peers),
```

```
    NewSuccessor = successor(get_active(S#state.peers),Id),
    NewPeers = lists:keyreplace(Id,1,S#state.peers,{Id,Predecessor,
          setelement(1,Successors,NewSuccessor),Fingers}),
    S#state{peers = NewPeers};

next_state(S,_,{call,_,stop,[Id,_]}) ->
    {value,{Id,Predecessor,Successors,Fingers}} = lists:keysearch(Id,1,S#state.peers),
    NewPeers = lists:keyreplace(Id,1,S#state.peers,{Id,Predecessor,
          setelement(1,Successors,undefined),setelement(1,Fingers,undefined)}),
    S#state{peers = NewPeers};

next_state(S,_,{call,_,stabilize,[Id,_]}) ->
    {value,{Id,_,Successors,Fingers}} = lists:keysearch(Id,1,S#state.peers),
    NewPredecessor = predecessor(get_active(S#state.peers),Id),
    NewPeers = lists:keyreplace(Id,1,S#state.peers,{Id,NewPredecessor,Successors,Fingers}),
    S#state{peers = NewPeers};

next_state(S,_,{call,_,update_successors,[Id,_]}) ->
    {value,{Id,Predecessor,_,Fingers}} = lists:keysearch(Id,1,S#state.peers),
    NewSuccessors = successors(get_active(S#state.peers),Id),
    NewPeers = lists:keyreplace(Id,1,S#state.peers,{Id,Predecessor,NewSuccessors,Fingers}),
    S#state{peers = NewPeers};

next_state(S,_,{call,_,update_fingers,[Id,_]}) ->
    {value,{Id,Predecessor,Successors,_}} = lists:keysearch(Id,1,S#state.peers),
    NewFingers = fingers(get_active(S#state.peers),Id),
    NewPeers = lists:keyreplace(Id,1,S#state.peers,{Id,Predecessor,Successors,NewFingers}),
    S#state{peers = NewPeers};

next_state(S,_,{call,_,lookup,_}) ->
   S.

%%====================================================================
% Internal functions
%%====================================================================

get_inactive(List) ->
   [Id || {Id,_,Successors,_} <- List, element(1,Successors) == undefined].

get_active(List) ->
   [Id || {Id,_,Successors,_} <- List, element(1,Successors) /= undefined].

all_inactive(List) ->
     lists:all(fun({_,_,Successors,_}) -> element(1,Successors) == undefined end, List).

not_standalone(List) ->
   length(get_active(List)) > 1.

active_and_inactive(List) ->
     (get_active(List) /= []) and (get_inactive(List) /= []).

wait(T) -> receive after T -> ok end.

successor(L,P) ->
   case (P == lists:last(L)) or (length(L) == 1) of
      true ->
         hd(L);
      false ->
         successor(L,P,length(L))
  end.

successor(L,_,1) ->
```

```
    lists:nth(2,L);

successor(L,P,N) ->
  case lists:nth(N-1,L) == P of
      true ->
        lists:nth(N,L);
      false ->
        successor(L,P,N-1)
  end.

predecessor(L,P) ->
   case (P == hd(L)) or (length(L) == 1) of
      true ->
        lists:last(L);
      false ->
        predecessor(L,P,length(L))
  end.

predecessor(L,_,2) ->
   lists:nth(1,L);

predecessor(L,P,N) ->
  case lists:nth(N,L) == P of
      true ->
        lists:nth(N-1,L);
      false ->
        predecessor(L,P,N-1)
  end.

successors(L,P) ->
   list_to_tuple(successors(L,P,?space_size)).

successors(L,P,1) ->
   [successor(L,P)];

successors(L,P,N) ->
   [successor(L,P)] ++ successors(L,successor(L,P),N-1).

fingers(L,P) ->
   list_to_tuple(fingers(L,P,?space_size)).

fingers(_,_,0) ->
   [];

fingers(L,P,N) ->
   Space = round(math:pow(2,?space_size)),
   Start = list_to_atom(integer_to_list((list_to_integer(atom_to_list(P)) +
           round(math:pow(2,N-1))) rem Space)),
   Finger = finger(L,Start),
   fingers(L,P,N-1) ++ [Finger].

finger(L,P) ->
   case P > lists:last(L) of
      true ->
        hd(L);
      false ->
        finger(L,P,length(L))
   end.

finger(L,_,1) ->
   hd(L);
```

```
finger(L,P,N) ->
  case (lists:nth(N-1,L) < P) and (P =< lists:nth(N,L)) of
      true ->
         lists:nth(N,L);
      false ->
         finger(L,P,N-1)
  end.
```

Appendix B: New Chord implementation

```
%%--------------------------------------------------------------------------------------------------------------
% File              : chord.erl
% Description       : Chord implementation
% Last modified : 30 March, 2007
%%--------------------------------------------------------------------------------------------------------------

-module(chord).

%%--------------------------------------------------------------------------------------------------------------
% External exports
%%--------------------------------------------------------------------------------------------------------------

-export([start/1,start/2,init/1,init/3,stop/1,
        stabilize/1,update_successors/1,update_fingers/1,lookup/2,
        client/1,client/2,exit/1]).

-import(erlang,[monitor/2,demonitor/1]).

%%--------------------------------------------------------------------------------------------------------------
% Definitions
%%--------------------------------------------------------------------------------------------------------------

-define(space_size,4).

%%--------------------------------------------------------------------------------------------------------------
% Records
%%--------------------------------------------------------------------------------------------------------------

-record(peer,{id,predecessor,successors = erlang:make_tuple(?space_size,undefined),
                        fingers = erlang:make_tuple(?space_size,undefined),data = []}).

%%==============================================================================
% External functions
%%==============================================================================

client(ID) ->
    start(ID),
    timer:apply_after(10,?MODULE,stabilize,[ID]),
    timer:apply_after(10,?MODULE,update_successors,[ID]),
    timer:apply_after(10,?MODULE,update_fingers,[ID]).

client(ID,Gate) ->
    start(ID,Gate),
    timer:apply_after(10,?MODULE,stabilize,[ID]),
    timer:apply_after(10,?MODULE,update_successors,[ID]),
    timer:apply_after(10,?MODULE,update_fingers,[ID]).

exit(ID) ->
    stop(ID).

%%--------------------------------------------------------------------------------------------------------------
% Function: start/1
% Descrip.: Start and register a Chord process
% Returns : bool()
%%--------------------------------------------------------------------------------------------------------------

start(ID) ->
    register(ID,spawn(?MODULE,init,[ID])).
```

```
%----------------------------------------------------------------------------------------------------------
% Function: start/2
% Descrip.: Start and register a Chord process via a known process
% Returns : bool()
%----------------------------------------------------------------------------------------------------------

start(ID,Gate) ->
   register(ID,spawn(?MODULE,init,[ID,Gate,self()])),
   receive
      {started,Predecessor,Successor} ->
         {ok,ID,Predecessor,Successor}
      end.

stop(Name) ->
  Pid = whereis(Name),
   exit(Pid,kill),
  unregister(Name).

init(ID) ->
   State = init_state(ID),
   process_flag(trap_exit,true),
   loop(State).

init(ID,Gate,ParentID) ->
   State = init_state(ID),
   process_flag(trap_exit,true),
   join(State,Gate,ParentID).

join(State,Gate,ParentID) ->
   ID = State#peer.id,
   Gate ! {request_neighbors,ID},
   receive
      {respond_neighbors,ID,Predecessor,Successor} ->
         monitor(process,Successor),
         monitor(process,Predecessor),
         ParentID ! {started,Predecessor,Successor},
         NewState = State#peer{predecessor = Predecessor,
                               successors = setelement(1,State#peer.successors,Successor),
                               fingers = setelement(1,State#peer.fingers,Successor)},
         loop(NewState)
   end.

%----------------------------------------------------------------------------------------------------------
% Function: stabilize/1
% Descrip.: Fixing successor/predecessor pointers
% Returns : tuple()
%----------------------------------------------------------------------------------------------------------

stabilize(Peer) ->
   Peer ! {stabilize,self()},
   receive
      {ID,Predecessor} ->
         {ID,Predecessor}
  end.

%----------------------------------------------------------------------------------------------------------
% Function: update_successors/1
% Descrip.: Refresh successor list
% Returns : atom()
%----------------------------------------------------------------------------------------------------------

update_successors(Peer) ->
```

```
    Peer ! {update_successors,self()},
    receive
        Successors ->
            Successors
    end.

%--------------------------------------------------------------------------------------------------------
% Function: update_fingers/1
% Descrip.: Refresh finger table
% Returns : false or tuple()
%--------------------------------------------------------------------------------------------------------

update_fingers(Peer) ->
    Peer ! {update_fingers,self()},
    receive
        {ID,Start,Finger} ->
            {ID,Start,Finger}
        after 10 ->
            false
    end.

%--------------------------------------------------------------------------------------------------------
% Function: lookup/2
% Descrip.: Find the node responsible for Key
% Returns : tuple(): {Key,Node}
%--------------------------------------------------------------------------------------------------------

lookup(Gate,Key) ->
    Gate ! {lookup,Key,self()},
    receive
        {lookup,Peer} ->
            {Key,Peer}
    end.

loop(State) ->

    ID = State#peer.id,
    Predecessor = State#peer.predecessor,
    Successors = State#peer.successors,
    Successor = element(1,Successors),

    receive

        {request_neighbors,AnotherPeer} ->
            case ID == Successor of
                true ->
                    NewSuccessor = AnotherPeer,
                    monitor(process,NewSuccessor),
                    AnotherPeer ! {respond_neighbors,AnotherPeer,ID,ID},
                    NewState = State#peer{successors = setelement(1,State#peer.successors,NewSuccessor),
                                          fingers = setelement(1,State#peer.fingers,NewSuccessor)},
                    loop(NewState);
                false ->
                    case is_element3(AnotherPeer,ID,Successor) of
                        true ->
                            NewSuccessor = AnotherPeer,
                            monitor(process,NewSuccessor),
                            AnotherPeer ! {respond_neighbors,AnotherPeer,ID,Successor},
                            NewState = State#peer{successors = setelement(1,State#peer.successors,NewSuccessor),
                                                  fingers = setelement(1,State#peer.fingers,NewSuccessor)},
                            loop(NewState);
                        false ->
```

```
                Successor ! {request_neighbors,AnotherPeer},
                loop(State)
            end
        end;

    {stabilize,ParentID} ->
        Successor ! {request_predecessor,ID,ParentID},
        loop(State);

    {request_predecessor,AnotherPeer,ParentID} ->
        AnotherPeer ! {respond_predecessor,Predecessor,ParentID},
        loop(State);

    {respond_predecessor,AnotherPeer,ParentID} ->
        case is_element(AnotherPeer,ID,Successor) and (whereis(AnotherPeer) /= undefined) of
            true ->
                NewSuccessor = AnotherPeer,
                monitor(process,NewSuccessor),
                NewState = State#peer{successors = setelement(1,State#peer.successors,NewSuccessor),
                                      fingers = setelement(1,State#peer.fingers,NewSuccessor)},
                AnotherPeer ! {notify_predecessor,ID,ParentID},
                loop(NewState);
            false ->
                Successor ! {notify_predecessor,ID,ParentID},
                loop(State)
        end;

    {notify_predecessor,AnotherPeer,ParentID} ->
        case (Predecessor == ID) or is_element(AnotherPeer,Predecessor,ID) of
            true ->
                NewPredecessor = AnotherPeer,
                ParentID ! {ID, NewPredecessor},
                monitor(process,NewPredecessor),
                case Successor == ID of
                    true ->
                        NewSuccessor = AnotherPeer,
                        monitor(process,NewSuccessor),
                        AnotherPeer ! {notify_predecessor,ID,ParentID},
                        NewState = State#peer{predecessor = NewPredecessor,
                                successors = setelement(1,State#peer.successors,NewSuccessor),
                                fingers = setelement(1,State#peer.fingers,NewSuccessor)},
                        loop(NewState);
                    false ->
                        NewState = State#peer{predecessor = NewPredecessor},
                        loop(NewState)
                end;
            false ->
                ParentID ! {ID, Predecessor},
                loop(State)
        end;

    {update_successors,ParentID} ->
        case Successor == ID of
            true ->
                ParentID ! Successors,
                loop(State);
            false ->
                Successor ! {request_successors,ID},
                receive
                    {respond_successors,NextSuccessors} ->
                        NewSuccessors = list_to_tuple([element(1,Successors)] ++
                                tuple_to_list(NextSuccessors) -- [element(?space_size,NextSuccessors)]),
```

```
                ParentID ! NewSuccessors,
                NewState = State#peer{successors = NewSuccessors},
                loop(NewState)
            end
    end;

{request_successors,AnotherPeer} ->
        AnotherPeer ! {respond_successors,State#peer.successors},
        loop(State);

{update_fingers,ParentID} ->
        Space = round(math:pow(2,?space_size)),
        Start = (list_to_integer(atom_to_list(State#peer.id)) +
        round(math:pow(2,random:uniform(?space_size) - 1))) rem Space,
        {_,Finger} = closest_preceding_finger(State,Start),
        cast(Finger,{request_finger,Start,ID,ParentID}),
        loop(State);

{request_finger,Start,AnotherPeer,ParentID} ->
    case Start == ID of
        true ->
            cast(AnotherPeer,{respond_finger,Start,ID,ParentID}),
            loop(State);
        false ->
            case is_element2(Start,ID,Successor) of
                true ->
                    cast(AnotherPeer,{respond_finger,Start,Successor,ParentID}),
                    loop(State);
                false ->
                    {_,Finger} = closest_preceding_finger(State,Start),
                    cast(Finger,{request_finger,Start,AnotherPeer,ParentID}),
                    loop(State)
            end
    end;

{respond_finger,Start,Finger,ParentID} ->
    Space = round(math:pow(2,?space_size)),
    Index = 1 + round(math:log((Start + Space - ID) rem Space)/math:log(2)),
    ParentID ! {ID,Start,Finger},
    NewState = State#peer{fingers = setelement(Index,State#peer.fingers,Finger)},
    loop(NewState);

{'DOWN',_,process,{Successor,_},_} ->
    NewSuccessor = next_successor(ID,State#peer.successors),
    monitor(process,Successor),
    NewState = State#peer{successors = setelement(1,State#peer.successors,NewSuccessor)},
    monitor(process,NewSuccessor),
    loop(NewState);

{'DOWN',_,process,{Predecessor,_},_} ->
    NewState = State#peer{predecessor = ID},
    loop(NewState);

{'DOWN',_,process,{_,_},_} ->
    loop(State);

{lookup,Key,ParentID} ->
    case ID == Successor of
        true ->
            ParentID ! {lookup,ID},
            loop(State);
        false ->
```

```
                case is_element3(Key,ID,Successor) of
                    true ->
                        ParentID ! {lookup,Successor},
                        loop(State);
                    false ->
                        Successor ! {lookup,Key,ParentID},
                        loop(State)
                end
        end

end.

%=====================================================================
% Internal functions
%=====================================================================

init_state(Pid) ->
    State = #peer{id = Pid, predecessor = Pid},
    State#peer{successors = setelement(1,State#peer.successors,Pid),
                        fingers = setelement(1,State#peer.fingers,Pid)}.

closest_preceding_finger(State,ID) ->
     closest_preceding_finger(State,ID,?space_size).

closest_preceding_finger(State,_,1) ->
    {1,element(1,State#peer.fingers)};

closest_preceding_finger(State,ID,N) ->
    Finger = element(N,State#peer.fingers),
    if
        (State#peer.id < Finger) and (Finger =< ID) ->
            {N,element(N,State#peer.fingers)};
        (State#peer.id > ID) and ((Finger > State#peer.id) or (Finger =< ID)) ->
            {N,element(N,State#peer.fingers)};
        true ->
            closest_preceding_finger(State,ID,N-1)
    end.

is_element(Item,Head,Tail) ->
    if
        (Head < Item) and (Item < Tail) ->
            true;
        (Head > Tail) and ((Item > Head) or (Item < Tail)) ->
            true;
        true ->
            false
    end.

is_element2(Item,Head,Tail) ->
     if
        (Head =< Item) and (Item < Tail) ->
            true;
        (Head > Tail) and ((Item >= Head) or (Item < Tail)) ->
            true;
        true ->
            false
    end.

is_element3(Item,Head,Tail) ->
    if
         (Head < Item) and (Item =< Tail) ->
              true;
```

```
        (Head > Tail) and ((Item > Head) or (Item =< Tail)) ->
              true;
        true ->
             false
   end.

cast(Name,Msg) ->
     catch Name ! Msg,
     ok.

next_successor(ID,Tuple) ->
    next_successor(ID,Tuple,2).

next_successor(ID,_,?space_size + 1) ->
    ID;

next_successor(ID,Tuple,N) ->
   Current = element(N,Tuple),
   case Current of
      undefined ->
           ID;
      _ ->
           case whereis(Current) of
               undefined ->
                  next_successor(ID,Tuple,N + 1);
               _ ->
                  Current
           end
   end.
```