Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 181 (2007) 35–47

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Verification of Peer-to-peer Algorithms: A Case Study

Rana Bakhshi[1]

School of Information and Communication Technology
Royal Institute of Technology
Kista, Sweden

Dilian Gurov[2]

School of Computer Science and Communication
Royal Institute of Technology
Stockholm, Sweden

Abstract

The problem of maintaining structured peer-to-peer (P2P) overlay networks in the presence of concurrent joins and failures of nodes is the subject of intensive research. The various algorithms underlying P2P systems are notoriously difficult to design and analyse. Thus, when verifying P2P algorithms, a real challenge is to find an adequate level of abstraction at which to model the algorithms and perform the verifications. In this paper, we propose an abstract model for structured P2P networks with ring topology. Our model is based on process algebra, which, with its well-developed theory, provides the right level of abstraction for the verification of many basic P2P algorithms. As a case study, we verify the correctness of the stabilization algorithm of Chord, one of the best-known P2P overlay networks. To show the correctness of the algorithm, we provide a specification and an implementation of the Chord system in process algebra and establish bisimulation equivalence between the two.

Keywords: Peer-to-peer systems, verification, process algebra.

1 Introduction

In recent years, great importance has been placed on distributed applications and P2P networks. Such systems are decentralized and highly dynamic, allowing an arbitrary number of nodes to join and leave the network. The essential operation in most P2P systems is the efficient location of data items, and the question whether a node is reachable in the network is a crucial problem. P2P network constructs

[1] Email:ranab@kth.se

[2] Email:dilian@nada.kth.se

doi:10.1016/j.entcs.2007.01.052

and maintains an overlay network with specific topology depending on the application requirements. Robust P2P systems produce a desired topology starting from assumed initial state. For large systems, careful design is needed to ensure the correctness of stabilization algorithms. An example of such a system is Chord [14], a simple robust structured P2P system that implements a Distributed Hash Table (DHT) abstraction [3]. A DHT maps keys (data identifiers) to the nodes of an overlay network and provides facilities for locating the current peer node responsible for a given node. Chord maintains its distributed state as nodes join and leave the system by executing the procedure called *stabilization algorithm*.

In the literature that describes such algorithms, e.g. the original Chord paper by Stoica *et al.* [15], correctness proofs are sketched at a high level of abstraction and tend to provide no operational semantics. On the other hand, model-checking techniques are not directly applicable to the verification of P2P systems as these systems are inherently dynamic and have infinite-state behaviour. Thus, a real challenge is to find a right level of abstraction at which to model the algorithms and perform the verification.

This paper summarizes the results of a formal verification of Chord's stabilization algorithm. It focuses on the *Pure Join Model* of the Chord protocol [14] where an arbitrary number of nodes can join the network and each node runs the stabilization algorithm. We analyse the correctness of the stabilization algorithm using process algebra. The process algebra chosen is the π-calculus [8,13], which is a natural language for modelling concurrent and distributed programs and particularly suitable for specifying mobile systems with dynamically changing communication topologies.

The main results of this paper are:

- An abstract formulation of dynamic networks with ring topology (Sect. 4).
- Modelling of both a specification and an implementation of the dynamic Chord system in terms of the π-calculus (Sect. 4).
- Formal proof of equivalence of the specification and the implementation using weak bisimulation relation, thus establishing the correctness of Chord's stabilization algorithm (Sect. 5).

Related Work.

All previous studies that model ring-based network with process algebra assume a fixed number of nodes in the network. A lookup algorithm of the DHT-based DKS system has been verified for a static model of the network using value-passing CCS [4]. Palamidessi [10] shows the leader election problem on a symmetric ring of processes with π-calculus with mixed choice. The paper makes an assumption about the existence of a special free outgoing channel for communication with the "external world". It also assumes the existence of a special subset from the set of names equipped with one-to-one mapping with the natural numbers to identify the individual processes in a network. Phillips *et al.* [11] take a similar approach to solve the same problem in the calculus of Mobile Ambients.

Other formal approaches, for instance the assertional proof method, has been used for the problem of concurrent maintenance of the ring topology in P2P networks [7]. Krishnamurthy *et al.* [6] present an analytical study of Chord under churn using a master-equation-based approach.

Note that a formal justification that the model, proposed in our work, conforms the actual algorithm is beyond the ambitions of the present paper. But, the relative simplicity of the algorithm, written in the syntax of the Erlang programming language, allows to relate both by simple inspection with a high level of confidence. The study by Noll and Roy [9] addresses the problem of mapping Erlang into the π-calculus.

2 Chord Protocol

In this section we briefly describe the Chord protocol, focusing on its stabilization algorithm. The details not relevant to this work, e.g. key's assignment to a node, are left out of the paper and can be found in Stoica *et al.* [14].

Chord is a self-organizing distributed P2P lookup protocol that provides support for one operation: given a key, it maps the key onto a node (Fig. 1). Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing.

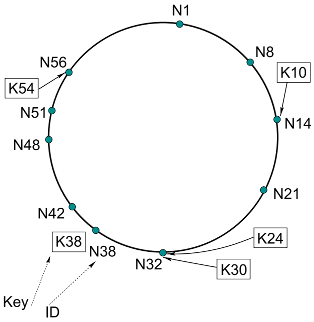

Fig. 1: Chord ring

Identifier space

In Chord, every node has an *n-bit identifier* (ID), which is assumed to be unique and belongs to the set called an *ID space*. Nodes' ID space forms a ring of size n, denoted with $\mathbb{N}_n$, where $\mathbb{N}_n := \{k_1, k_2, ..., k_n\} \subseteq \mathbb{N}$, $\mathbb{N}$ is a set of natural numbers. The values sent and received are terms ranged over finite set of indices $\mathcal{I} \subseteq \mathbb{N}_n$. We let $\boxplus$ and $\boxminus$ be addition and subtraction modulo n ranged over $\mathbb{N}$ and the result is always strictly less than the modulus.

When a new node joins the network, it should be placed between two nodes with proper IDs in the ring. The two basic neighbours that a node has are its predecessor and successor, chosen according to the partial order on $\mathcal{I}$. A node is called a *successor* of the node with ID x in the set $\hat{m}$ if it is the first node succeeding x in the set $\hat{m}$ and is defined as: $\text{succ}(x, \hat{m}) = \{y \in \hat{m} | (y \boxminus x) = \min\{(z \boxminus x) | z \in \hat{m}\} \wedge \hat{m} \subseteq \mathcal{I}\}$. The function $\text{succ}\colon \mathcal{I} \times \mathcal{P}(\mathcal{I}) \to \mathcal{I}$ is well-defined, as $z \boxminus x = y \boxminus x$ iff $z \boxminus y = 0$; $\mathcal{P}(\mathcal{I})$ is a powerset of $\mathcal{I}$. A *predecessor* of the node with ID x in the set $\hat{m}$ is defined as: $\text{pred}(x, \hat{m}) = \{y \in \hat{m} | (y \boxminus x) = \max\{(z \boxminus x) | z \in \hat{m}\} \wedge \hat{m} \subseteq \mathcal{I}\} \cup \{x \notin \hat{m} \mid \bot\}$. We use $\hat{m} \oplus x$ to denote union $\hat{m} \cup \{x\}$.

The Stabilization Algorithm.

In order to assert that lookups execute correctly as the set of participating nodes changes, Chord must ensure that each node's successor pointer is up to date. It does this using the *stabilization algorithm*, presented in Fig. 2.

Definition 2.1 A Chord ring is **stable** if $predecessor(successor(u)) = u$ for all nodes u, and there is no node v such that $u < v < successor(u)$.

The *Pure Join model* of the Chord protocol assumes the absence of nodes' failure. To simplify the network representation this model doesn't include successor lists and finger tables considered in the general model of the protocol. The stabilization scheme guarantees nodes to join a ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins.

```
%new node joins a Chord ring containing node n'
join() ->
 n'! {"find_successor",n}
 receive
     {"find_successor",n'} ->
                succ := n'; pred := nil;
 end.

%in parallel with

loop
  receive
    {"find_successor",n'}  ->
        if n'∈(n,succ] ->
               n'! {"find_successor",succ}
           n'∉(n,succ] ->
               succ! {"find_successor",n'}
  end
end.

%stabilization algorithm
stab() ->
 succ! {"req_predecessor",n}
 receive
     {"resp_predecessor",n'} ->
        if n'∈(n,succ) ->
               succ := n';
        succ! {"notify",n}
 end.

%in parallel with

loop
  receive
    {"req_predecessor",n'} ->
        n'!{"resp_predecessor",pred}
    {"notify",n'}  ->
        if pred = nil ∨ n'∈(pred,n) ->
               pred :=n';
  end
end.
```

Fig. 2. Stabilization on Join algorithm

Although the stabilization algorithm of Chord is presented in "RPC-style" in the original paper [14], any actual implementation would use asynchronous message-passing as it is the natural communication discipline for P2P systems [5]. This is reflected in our version of the stabilization algorithm, given in Fig. 2.

The algorithm is presented in the syntax of the Erlang programming language [1]. For a formal semantics of Erlang, see e.g. [12]. In Erlang, the communication between processes is asynchronous by means of process message queues ("mailboxes"). Briefly, n, n', pred, succ are node IDs; primitive $\rightarrow$ denotes "then" like notation; $\mathcal{I}!$ defines sending event to the process $\mathcal{I}$; assignment is denoted as :=; `receive` $\mathcal{X}$ `end` inspects the process incoming mailbox and delivers the first $\mathcal{X}$ pattern-matching element; `loop` $\mathcal{X}$ `end` executes $\mathcal{X}$ in loop. $\{$"$\mathcal{M}$", $\mathcal{I}\}$ represents a message of type $\mathcal{M}$, sent by process with ID $\mathcal{I}$.

When node n first starts, it runs `join(n')`, like the one shown in Fig. 2, where n' is any known Chord node. The `join()` function asks n' to find the immediate successor of n, but does not make the rest of the network aware of n. All nodes execute code, given in lines 8–16 in Fig. 2, for appropriated response.

Each node in the system executes `stab()` periodically. When node n runs `stab()` (see Fig. 2), it asks its successor for the successor's predecessor p, and decides whether p should be n's successor instead. This would be a case if node p joined the system. Also, `stab()` notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n.

This stabilization algorithm [14] guarantees the convergence of the Chord ring to a stable configuration, i.e. a ring topology, in spite of the concurrent joins if the system starts in stable configuration (Def. 2.1).

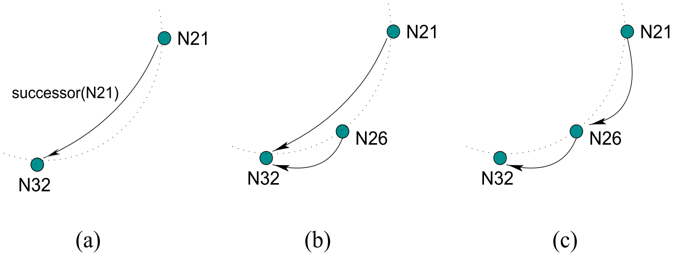

Fig. 3. Illustration of stabilization on join

Figure 3 illustrates an example of the join operation. Node 26 joins the system between nodes 21 and 32. The arcs represent the successor relationship. (a) Initially, node 21 points to node 32; (b) node 26, by executing `join()`, asks any known node to find its successor (i.e., 32) and points to it; (c) `stab()` updates the successor of node 21 to node 26.

3 π - calculus

As the Chord system is dynamic, i.e. nodes can freely join the network, our choice of modelling formalism is the π-calculus [8,13]. It is based on the concepts of processes and names. Communication between processes takes place on names (channels). A process Q that offers an output of the value v on the channel p, i.e. $\overline{p}\langle v\rangle.Q$, may synchronise with a parallel process Q' that attempts to read from channel p, i.e. $p(z).Q'$. Names sent as values can be used later as channels for communication. This feature allows processes to establish connections dynamically during computation.

This section briefly reviews the syntax and the operational semantics of the π-calculus. For more details refer to Milner [8] and Sangiorgi and Walker [13]. The section also draws inspiration from the prior work of Borgström *et al.* [4] on the static case of structured P2P systems.

u, v	$::=$	o, p, x	names
	$\mid$	in, id	channels
	$\mid$	$\perp$	undefined value
	$\mid$	k_i, k_j	integers, IDs
	$\mid$	$\mathrm{succ}(x, m)$	successor
	$\mid$	$\mathrm{pred}(x, m)$	predecessor
e, e', e''	$::=$	u	expressions
ϕ, ψ	$::=$	$e = e'$	boolean tests
	$\mid$	$e \in (e', e'']$	interval check
m, m', m''			sets, subsets
π	$::=$	$\tau \mid \bar{p}\langle\overrightarrow{v}\rangle \mid p(\overrightarrow{v})$	prefix
	$\mid$	$(\overrightarrow{v}).P$	abstractions
	$\mid$	$\overrightarrow{u}\langle\overrightarrow{v}\rangle.P$	concretions
Q, R	$::=$		processes
	$\mid$	M	summation
	$\mid$	$(Q \mid R)$	parallel composition
	$\mid$	$(\nu\overrightarrow{p})\ Q$	restriction
	$\mid$	if ψ then Q else R	if statement
	$\mid$	$!Q$	replication
	$\mid$	$Q\langle\overrightarrow{v}\rangle$	process constant
M, M'	$::=$	$\mathbf{0}$	inaction
	$\mid$	$\pi.Q$	process action
	$\mid$	$M + M'$	choice

Fig. 4. Syntax

Our model and verification of the Chord protocol's stabilization scheme use a variant of the extended polyadic π-calculus (Fig. 3).

Operational Semantics.

The operational semantics of the π-calculus is defined by the commitment rules [8,13]:

$$\text{(R-RCT-L)}\,\frac{-}{(p(z).Q + M)|(\overline{p}\langle v\rangle.Q' + M') \xrightarrow{\tau} \{v/z\}Q|Q'} \tag{1}$$

$$\text{(R-TAU)}\,\frac{-}{\tau.Q + M \xrightarrow{\tau} Q} \tag{2}$$

$$\text{(R-PAR-L)}\frac{Q \xrightarrow{\pi} Q'}{Q|R \xrightarrow{\pi} Q'|R} \tag{3}$$

$$\text{(R-RES)}\frac{Q \xrightarrow{\pi} Q'}{(\nu p)Q \xrightarrow{\pi} (\nu p)Q'} \tag{4}$$

The symmetric versions of (1), (3) are omitted. Evaluation of expressions is a function $[\![\cdot]\!] : \mathcal{E}^\pi \rightarrow \mathbb{N}$, where $\mathcal{E}^\pi$ is a set of π-calculus expressions (Fig. 3):

$$[\![e]\!] = \{k_i, \text{ if } e = k_i; \bot, \text{ otherwise}\}$$

Predicate $e_b(\cdot)$ performs a boolean check:

$$e_b(e_1 = e_2) \text{ is true iff } [\![e_1]\!] = [\![e_2]\!] \tag{5}$$

$$\begin{gathered} e_b(e_1 \in [e_2, e_3]) \text{ is true iff } [\![e_i]\!] = k_i \in \mathcal{I}, \\ \text{where } i = \overline{1,3} \text{ and } 0 \leq k_1 \boxminus k_2 \leq k_3 \boxminus k_2 \end{gathered} \tag{6}$$

To evaluate the expressions of *if* statements, π-calculus is extended with the reduction relation $>$, satisfying the rules:

$$\text{if } \psi \text{ then } Q \text{ else } Q' > Q, \text{ if } e_b(\psi) \tag{7}$$

$$\text{if } \psi \text{ then } Q \text{ else } Q' > Q', \text{ if } \neg e_b(\psi) \tag{8}$$

and a corresponding commitment rule:

$$\text{(R-RED)}\frac{Q > Q' \; Q' \xrightarrow{\pi} Q''}{Q \xrightarrow{\pi} Q''} \tag{9}$$

Bisimulation.

In the π-calculus, the standard equivalence relation on processes is *bisimulation* [8,13]. Intuitively, two systems are *bisimilar* if they can stepwise match each other's actions. The difference between *strong* and *weak* versions is the view on silent actions, τ. Let $\mathcal{P}^\pi$ be a set of π-calculus process expressions (Fig. 3).

Definition 3.1 For any prefix π, given in Fig. 3, the relations $\Rightarrow$ and $\overset{\pi}{\Rightarrow}$ are defined as follows:

(i) $P \Rightarrow Q$ means that there is a sequence of zero or more reactions $P \rightarrow ... \rightarrow Q$. Formally, $\Rightarrow \overset{def}{=} \rightarrow^*$, the transitive reflexive closure of $\rightarrow$.

(ii) Let $\pi = p_1...p_n$. $P \xrightarrow{\pi} Q$ means $P \Rightarrow \xrightarrow{p_1} P_1 ... \Rightarrow \xrightarrow{p_n} P_n \Rightarrow Q$. Formally,$\xrightarrow{\pi} \overset{def}{=} \Rightarrow \xrightarrow{p_1} \Rightarrow ... \Rightarrow \xrightarrow{p_n} \Rightarrow$.

Definition 3.2 A binary relation $\mathcal{S}$ over set $\mathcal{P}^\pi$ is a *strong simulation* if, whenever $P\mathcal{S}Q$, if $P \xrightarrow{\pi} P'$ then $\exists Q'.\ Q \xrightarrow{\pi} Q'$ and $P'\mathcal{S}Q'$.

If both $\mathcal{S}$ and its converse are strong simulations then $\mathcal{S}$ is a *strong bisimulation* and is denoted by $\sim$.

Definition 3.3 A binary relation $\mathcal{S}$ over set $\mathcal{P}^{\pi}$ is a *weak simulation* if, whenever $P\mathcal{S}Q$,

if $P \Rightarrow P'$ then $\exists Q'.\ Q \Rightarrow Q'$ and $P'\mathcal{S}Q'$;

if $P \overset{p\langle\overline{u}\rangle}{\Longrightarrow} P'$ then $\exists Q'.\ Q \overset{p\langle\overline{u}\rangle}{\Longrightarrow} Q'$ and $P'\mathcal{S}Q'$;

if $P \overset{\overline{p}}{\Rightarrow} P'$ then $\exists Q'.\ Q \overset{\overline{p}}{\Rightarrow} Q'$ and $P'\mathcal{S}Q'$.

If both $\mathcal{S}$ and its converse are weak simulations then $\mathcal{S}$ is a *weak bisimulation* and is denoted by $\approx$.

4 Modelling

We now describe a formal model for the specification and implementation of the Chord protocol. The dynamic model of the system should clearly define the topology and express the joining of nodes to the system. Note that all channels are unidirectional and single purpose (i.e. each type of channel is used for one type of a message, shown in Fig. 7), as it is suggested for the base π-calculus.

Specification.

We specify the Chord protocol as a collection of nodes in a network, forming a ring topology and growing in size.

In general, nodes can join the network in arbitrary order. Since the internal structure of the system is hidden from the observer, a major problem is how to specify the behaviour of a ring. The approach we propose is to add a token ring protocol, i.e. an abstract token to be passed clockwise along the ring, on top of the Chord protocol.

Process *Ring* represents a ring of nodes. Communication between the nodes is not specified; thus, internal links are restricted and unobservable. The only visible actions are the outputs on ports specific to every node, representing its ID (Fig. 5).

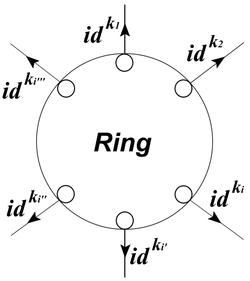

Fig. 5: Process *Ring*

$$
\begin{aligned}
Ring(k_i, m) \triangleq\ & \tau.\overline{id^{k_i}}.Ring\langle \mathrm{succ}(k_i, m), m\rangle \\
& + \sum_{k_j \in \mathcal{I}\setminus m} \tau.Ring\langle k_i, m \oplus k_j\rangle
\end{aligned} \tag{10}
$$

Here m represents a set of identifiers of the nodes already in the system; id^{k_i} is an output port performed only by node k_i and, therefore, denotes a node in an abstract way; $\mathrm{succ}(k_i, m)$ is the successor of node k_i from the set m; $\mathcal{I}$ is finite set of indices.

The *Ring* behaviour is expressed by the following non-deterministic choice:

```
%%new node join a Chord ring containing node n'
join() ->
 n'! {"find_successor",n}
 receive
     {"find_successor",n'} ->
                succ := n'; pred := nil;
 end.

%in parallel with

loop
  receive
    {"find_successor",n'}  ->
        if n'∈(n,succ] ->
               n'! {"find_successor",succ}
           n'∉(n,succ] ->
               succ! {"find_successor",n'}
  end
end.

%%stabilization algorithm
stab() ->
 succ! {"req_predecessor",n}
 receive
     {"resp_predecessor",n'} ->
        if n'∈(n,succ) ->
               succ := n';
               succ! {"notify",n}
        if (n'∉(n,succ) ∧ (n'≠n)) ->
               succ! {"notify",n}
 end.

%in parallel with

loop
  receive
    {"req_predecessor",n'} ->
        n'!{"resp_predecessor",pred}
    {"notify",n'}  ->
        if pred = nil ∨ n'∈(pred,n) ->
               pred :=n';
               if (pred = n) ∧ (succ = n) ->
                      succ := n';
                      succ!{"notify",n}
  end
end.
```

Fig. 6. Modified Stabilization algorithm

- A new node can join the ring allowing the network to grow by adding its ID to the set m. The procedure of acquiring the neighbours links is performed by τ. Here, we do not restrict the order, in which the nodes join the network, i.e. $\forall k_j \in \mathcal{I}\backslash m$.
- A node performs an output on its channel $(\overline{id^{k_i}})$, preceded by τ, and passes the token to the closest neighbour, its successor. This forms a cycle of sequential outputs, where each successor is waiting for its predecessor to be enabled.

Implementation.

Chord is implemented as a product of concurrent mobile processes - one process per node, where all request/responses are handled according to the algorithm, given in Fig. 2.

As with our specification, we extend the Chord protocol's implementation by adding a token ring port. This extension does not affect the stabilization algorithm as the reaction on this port doesn't cause the reaction on the algorithm's ports, i.e. in_1^x, in_2^x, in_3^x, in_4^x.

π-calculus uses synchronous communication, implying that a deadlock or erroneous behaviour may occur under some circumstances. This includes the case when a node tries to contact itself. Therefore, we have made two modifications on the original algorithm, shown in Fig. 2, to handle the case with one and two nodes in the network. The modified version of the stabilization algorithm appears in Fig. 6.

Process P is a node that is not in the network but may join it:

$$P(\overrightarrow{in^{k_j}}, in_4^{k_i}) \triangleq (\nu\ \overrightarrow{in^{k_j}}) \left(\overline{in_4^{k_i}}\langle\overrightarrow{in^{k_j}}\rangle.in_4^{k_j}(\overrightarrow{in^{z}}).A\langle id^{k_j}, \overrightarrow{in^{k_j}}, \overrightarrow{in^{k_z}}, \overrightarrow{\perp}\rangle\right) \tag{11}$$

Here k_j is the ID of a node that wishes to connect to the network and is not yet part of the ring; $in_4^{k_i}$ is the "find_successor" port of an arbitrary node in the Chord ring.

Process A, illustrated in Fig. 7, defines a network node with the encoded stabilization algorithm (Fig. 6). It stores information about itself, its successor and its predecessor:

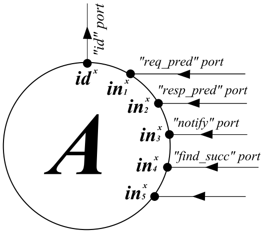

Fig. 7: Process A

$$\begin{aligned}
A(id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}) \triangleq\ & \overline{in_1^s}\langle\overrightarrow{in^o}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle \\
& + in_1^o(\overrightarrow{in^z}).\overline{in_2^z}\langle\overrightarrow{in^p}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^z}, \overrightarrow{in^p}\rangle \\
& + in_2^o(\overrightarrow{in^z}).(\text{if } z \in (o,s) \text{ then } \overline{in_3^z}\langle\overrightarrow{in^o}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^z}, \overrightarrow{in^p}\rangle \\
& \qquad\qquad \text{else } (\text{if } z = o \text{ then } A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle \\
& \qquad\qquad\qquad \text{else } \overline{in_3^s}\langle\overrightarrow{in^o}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle)) \\
& + in_3^o(\overrightarrow{in^z}).(\text{if } p = \perp \vee z \in (p,o) \\
& \quad \text{then } (\text{if } (p = o) \wedge (s = o) \text{ then } \overline{in_3^z}\langle\overrightarrow{in^o}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^z}, \overrightarrow{in^z}\rangle \\
& \qquad\qquad \text{else } A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^z}\rangle) \\
& \quad \text{else } A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle) \\
& + in_4^o(\overrightarrow{in^z}).(\text{if } z \in (o,s] \text{ then } \overline{in_4^z}\langle\overrightarrow{in^s}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle \\
& \qquad\qquad \text{else } \overline{in_4^s}\langle\overrightarrow{in^z}\rangle.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle) \\
& + in_5^o.\overline{id^o}.A\langle id^o, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}\rangle | \overline{in_5^s}.\mathbf{0}
\end{aligned} \tag{12}$$

where: $\overrightarrow{in^x} = \{x, in_1^x, in_2^x, in_3^x, in_4^x, in_5^x\}$.

Here o is ID of node, id^o - its special output port, $\overrightarrow{in^o}$ - ID and listening (in-) ports of node; $\overrightarrow{in^s}$ - ID and in-ports for successor, $\overrightarrow{in^p}$ - ID and in-ports for predecessor.

Node A uses in- ports of its neighbours to send the messages. These links are acquired by performing the following actions (see (12) and Fig. 6):

- Wait for a "find successor" message from the new node ($in_4^o(\overrightarrow{in^z})$), get node's ID and in-ports; check if the successor of the new node is the successor and, if true, return the successor's ID and in-ports to a new node ($\overline{in_4^z}\langle\overrightarrow{in^s}\rangle$), or if false, pass the information about the new node around the circle to the successor ($\overline{in_4^s}\langle\overrightarrow{in^z}\rangle$). This fragment is presented by lines 10–18 in Fig. 6.

- Receive a "request predecessor" message ($in_1^o(\overrightarrow{in^z})$) from another node, put the predecessor information into a message and send this message back ($\overline{in_2^z}\langle\overrightarrow{in^p}\rangle$). For the code details, see lines 36–37 of Fig. 6.
- Listen for the "response predecessor" message ($in_2^o(\overrightarrow{in^z})$), check if the condition $z \in (o, s)$ is satisfied, notify a sender so that it becomes the node's predecessor by sending its ID and in-ports and updating the successor information. If the check fails then confirm the successor to be its predecessor only if $z = o$. This corresponds to lines 24–27 in Fig. 6.
- Get notification from another node ($in_3^o(\overrightarrow{in^z})$) with a node's ID and in-ports, check the condition $p = \perp \vee z \in (p, o)$ - if true, update the predecessor information; in addition, update the successor's information and send the notification to it if $p = o = s$; if false, ignore the notification. For the code details, refer to lines 38–43 of Fig. 6.
- Request information from the successor by sending a "request predecessor" message ($\overline{in_1^s}\langle\overrightarrow{in^o}\rangle$). This is done according to line 22 of Fig. 6.
- Receive a token (in_5^o), release another one for the successor ($\overline{in_5^s}.\mathbf{0}$) and signal on special port ($\overline{id^o}$).

Process *Impl* implements Chord protocol and consists of collection of nodes A, nodes P ready to join a network and a token, all running in parallel:

$$\begin{aligned} Impl(k_i, m) \triangleq\ & (\nu \overrightarrow{in^{k_j}}, \overrightarrow{in^{k_i}}, \overrightarrow{in^{k_l}})((\overline{in_5^{k_i}}.\mathbf{0})|(\prod_{k_j \in \mathcal{I} \backslash m} \sum_{k_v \in m'} P\langle \overrightarrow{in^{k_j}}, in_4^{k_v}\rangle)| \\ & | \prod_{k_{i'} \in m'' \subset m} A\langle id^{k_{i'}}, \overrightarrow{in^{k_{i'}}}, \overrightarrow{s^{k_{i'}}}, \overrightarrow{p^{k_{i'}}}\rangle| \prod_{k_l \in m' \subseteq m} A\langle id^{k_l}, \overrightarrow{in^{k_l}}, \overrightarrow{s^{k_l}}, \overrightarrow{p^{k_l}}\rangle)) \end{aligned} \tag{13}$$

where $k_j \in \mathcal{I}\backslash m$ and $k_i \in m'$; $\overrightarrow{s^x} = \overrightarrow{in^{\mathrm{succ}(x,m)}}$ and $\overrightarrow{p^x} = \overrightarrow{in^{\mathrm{pred}(x,m)}}$; m'' is a set that contains all nodes that are connected but not settled in the Chord ring, i.e. haven't acquired the links; m' – a set of the nodes settled in the Chord ring, i.e. that have both neighbours. Note that m'' can be empty when all nodes are placed in the Chord ring and $m = m' \oplus m''$.

5 Verification

We now present the results of a formal verification of Chord's stabilization algorithm under the previously made assumptions. Verification gives us the confidence that the stabilization algorithm will eventually fix the immediate successor of each node (as we consider a *Pure Join model*) and the network will eventually form a ring topology again. Intuitively, the Chord ring becomes stable if all nodes will get correct information about their predecessors, i.e. a node will get both right and left neighbours.

We verify the correctness of the implementation with respect to the specification by establishing their behavioural equivalence. If the two models are bisimilar then the algorithm works correctly, i.e. eventually yields a ring topology encompassing the new nodes that join the network.

The proof technique we use to show the specification (10) and implementation (13) are (weakly) bisimilar is to prove that they are solutions of the same guarded systems of equations, and then to appeal to the Unique-Solution Theorem [8,13] for the extended π-calculus.

Theorem 5.1 *Let Ring and Impl be defined as above. Then*

$$Ring \approx Impl$$

The theorem is proved by showing that the specification (10) and the implementation (13) solve the following system of guarded equations:

$$X(k_i, m) \approx \tau.\overline{id^{k_i}}.X\langle \mathrm{succ}(k_i, m), m\rangle + \sum_{k_j \in \mathcal{I}\setminus m} \tau.X\langle k_i, m \oplus k_j\rangle \tag{14}$$

where $k_i \in m$, $k_j \in \mathcal{I}\setminus m$.

More details of the proof appear in the full version of this paper [2].

6 Conclusion

In this paper, we introduce an abstract model for structured P2P networks with a ring topology. Our choice of modelling formalism is the π-calculus, a calculus of mobile processes, and is based on the observation that P2P systems are of a highly dynamic nature, allowing nodes to join and leave the network at any time. We verify the correctness of Chord's stabilization algorithm by establishing weak bisimulation between the specification of Chord as a ring network and the implementation of the stabilization algorithm, both modelled in the π-calculus.

The present case study shows that the π-calculus offers a suitable theory for the verification of relatively simple P2P algorithms. It extends previous results on the correctness of a lookup algorithm of another P2P system for the static case without node joins and failures [4].

Future work is needed to include other aspects of P2P networks, including presence of failures in the network, adding finger tables and successors lists, etc. In addition, the π-calculus models can be formalized and the bisimulation proof can be carried out in a theorem proving system such as Isabelle/HOL.

References

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

[2] R. Bakhshi and D. Gurov. Verification of Peer-to-peer Algorithms: A Case Study. Technical report, ICT, May 2006. http://www.nada.kth.se/~dilian/Papers/chord.ps.gz.

[3] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, February 2003.

[4] Johannes Borgström, Uwe Nestmann, Luc Onana Alima, and Dilian Gurov. Verifying a Structured Peer-to-Peer Overlay Network: The Static Case. In *Global Computing*, pages 250–265, 2004.

[5] Geoffrey Fox. Message Passing: From Parallel Computing to the Grid. *Issue of Computing in Science and Engineering. IEEE Distributed Systems Online*, 3(8):70–73, September/October 2002.

[6] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A statistical theory of chord under churn. In *4th International Workshop on Peer-To-Peer Systems*, Ithaca, New York, USA, February 2005.

[7] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Brief announcement: concurrent maintenance of rings. In Soma Chaudhuri and Shay Kutten, editors, *PODC*, page 376. ACM, 2004.

[8] Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[9] Thomas Noll and Chanchal Kumar Roy. Modeling Erlang in the π-calculus. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 72–77, New York, NY, USA, 2005. ACM Press.

[10] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Symposium on Principles of Programming Languages*, pages 256–265, 1997.

[11] I.C.C. Phillips and M.G. Vigliotti. Leader election in rings of ambient processes. In *Proceedings of Express: Workshop on Expressiveness in Concurrency, London, August 2004*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 185–199. Elsevier, 2005.

[12] Lars–Åke Fredlund. *Framework for Reasoning about Erlang Code.* PhD thesis, Royal Institute of Technology, Kista, Sweden, August 2001.

[13] Davide Sangiorgi and David Walker. *π-Calculus: A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA, 2001.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[15] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical report, MIT, January 2002.