Formalization and Concretization of Ordered Networks

Laurence Rideau, Bernard P. Serpette, Cédric Tedeschi

Formalization and Concretization of Ordered Networks

Laurence Rideau, Bernard Serpette, Cédric Tedeschi

RESEARCH REPORT
N° 8172
December 2012
Project-Teams Indes, Marelle, Myriads

ISSN 0249-6399   ISRN INRIA/RR--8172--FR+ENG

Formalization and Concretization of Ordered Networks

Laurence Rideau, Bernard Serpette, Cédric Tedeschi

Project-Teams Indes, Marelle, Myriads

Research Report n° 8172 — December 2012 — 22 pages

Abstract: Overlay networks have been extensively studied as a solution to the dynamic nature, scale and heterogeneity of large computing platforms, and are a fundamental layers of most existing peer-to-peer networks. The basic mechanism offered by an overlay network, is *routing*, *i.e.*, the mechanism enabling the delivery of messages from any node to any other node in the network. On top of routing are built crucial functionalities of peer-to-peer networks, such as networks maintenance (nodes joining and leaving the network) and information distribution and retrieval.

Over the years, different topologies and routing mechanisms have been proposed in literature. However, there is a lack of formal works unifying these different designs and establishing their correctness.

This paper proposes a formal common basis, partially validated with the Coq theorem prover, with the nice property of only requiring the definition of a total order on the nodes. We investigate how such a basic design can be used to build deadlock/livelock-free algorithms for routing, node insertion, and node deletion in the fault-free environment.

The genericity of our design is then explored through the construction of orders on nodes corresponding to different topologies commonly encountered in the peer-to-peer domain. To validate the methodology proposed, a simulator tool was developed. This tool is able, given the definition of an order and the definition of shortcuts, to simulate the corresponding overlay network and to explore its performance.

Key-words: Formalization, peer-to-peer systems, routing

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Formalisation et concrétisations de réseaux ordonnés

Résumé :

Les réseaux d'overlays ont été proposés comme une solution à la nature hétérogène et dynamique des plates-formes de calcul à large échelle. Ils sont une couche fondamentale de beaucoup de réseaux pair-à-pair existants. Le mécanisme de base offert par un réseau d'overlay est le routage, c'est-à-dire le mécanisme permettant l'acheminement d'un message depuis n'importe quel nœud du réseau vers n'importe quel autre. Au-dessus du routage, on peut construire des fonctionnalités telles que la maintenance de la connexité du réseau (en présence de dynamicité) et la distribution et la recherche d'information.

Différentes topologies et mécanismes de routage ont été proposés dans la littérature. Toutefois, peu de travaux plus formels unifiant ces différentes propositions, et établissant plus formellement leur correction ont été proposés.

Ce rapport présente une base formelle commune, partiellement validée avec l'assistant de preuve Coq. En particulier, la construction proposée ne suppose que la présence d'un ordre total sur les nœuds. Nous montrons comment cette seule hypothèse peut être suffisante pour construire des algorithmes sans *deadlocks* ni *livelocks* pour router, insérer des nœuds, et supprimer des nœuds dans un environnement non sujet aux fautes.

La généricité de notre approche est explorée à travers la construction d'ordres au-dessus de topologies fréquemment rencontrées dans les systèmes pair-à-pair. Afin de valider la méthode proposée, un outil de simulation a été développé. Ce simulateur prend en entrée la définition d'un ordre et des raccourcis du réseau et rend un ensemble d'indicateurs de performances de la topologie ainsi construite.

Mots-clés : Formalisation, systèmes pair-à-pair, routage

Contents

1 Introduction

Overlay networks, whose primary advantage is to abstract out the difficulties of physical routing, is a crucial element of the leveraging of the emerging distributed systems, which are infinitely large and heterogeneous. Peer-to-peer systems make an extensive use of it [16].

A fundamental building block characterizing an overlay network is its *routing* mechanism. Routing is meant to deliver a message emitted at any node to any other node in the overlay. An important requirement is that this routing process should remain correct, in spite of nodes joining and leaving the networks, making the overlay grow and shrink over time.

A class of overlay networks broadly used over the years, is based on oriented rings [21, 18]. A lot of work went into enhancing such overlays with different optimizations and extensions, related to both functional and performance issues. Such examples include load balancing [12] or the support for complex queries [20]. Besides ring overlay networks, different topologies have been proposed, such as cartesian spaces [17] and hypercubes [19]. While the routing processes in these overlays all typically grows logarithmically with the size of the network, their design significantly differs.

Each overlay follows a particular topology and routing mechanism; each overlay has its own way to define *shortcuts*, *i.e.,* a set of distant links, maintained by each node in a routing table, for the routing efficiency. Although these overlays share a common goal (physical network abstraction and efficient message delivery) and common features (routing, node insertion, node deletion), very few works attempted to give them a common formal basis.

1.1 Existing works

We can find a lot of recent works aiming at maintaining overlay networks in faulty or adversarial settings [7, 5, 8, 10, 11, 2, 13]. Hayes *et al.* proposed the Forgiving Tree [7], a distributed structure that maintains, under periodic adversarial deletion of one of its node, strong guarantees on its diameter and degree. Upon each adversarial deletion, both the time needed to recover and the number of messages sent are constant. The authors have extended their work to adversarial insertions of nodes in [8], but still with one adversarial move at a time. In [2], a spanning tree maintenance algorithm is proposed, also providing strong guarantees about the degree and the diameter, but under an infinite arrival model with bounded concurrency (the number of concurrently active nodes at a given time is bounded). However, all these algorithms assume that the processes are reliable.

A recent series of work proposed self-stabilizing overlay structures. In [5], a self-stabilizing skip list is proposed. In [11], a variant of the skip graph distributed structure is made self-stabilizing, with the main advantage of achieving a sublinear convergence time, which represents an improvement over the earlier self-stabilizing dynamic network structures. In [10], the same authors came up with a similar result for Delaunay graphs. Recently, they proposed a self-stabilizing version of the famous Chord protocol [13].

All these works are based on repair mechanisms ensuring a quick recovery after a set of transient faults or attacks. Only few works have the goal of formally establishing the full continuity of the basic features of the overlay (routing, insertion, deletion) in a fault-free, non-adversarial setting. This paper follows this path, as maintaining the topology in the fault-free environments appears to be, in itself, a non trivial task. Also, as argued in [14], which exposes some formally-proved results for ring-based networks, and thus constitutes our closest related works, the fault-free setting remains a very relevant area to be explored, as the design of fault-tolerant protocols could naturally build upon formally proven results in fault-free environments. Note that, in the work in [14], the protocols proposed still suffer from livelocks. We can find a CCS formalization/verification of the lookup process in [4], but the dynamic insertion and deletion of nodes were not considered. In [15], Lynch *et al.* propose an interesting approach for ensuring atomic data access in the Chord overlay network. However, with such a protocol, there is no insurance that some message will not be sent to a node that already left the network. Our approach aims also at suppressing this drawback.

1.2 Contributions of this report

Our goal is to extract a minimal formal basis on top of which can be constructed and compared different overlays. Our contribution presents two facets. (1) We give, based on the minimal requirement of a total order over the nodes, formal generic deadlock/livelock-free algorithms for routing, inserting and deleting nodes. For the sake of correctness, this specification was tested with the Coq theorem prover [3]. (2) We show how this generic formalization can be used to build different topologies, subject to a dedicated order and a way to define shortcuts. To validate and illustrate the methodology, we finally discuss results obtained with a simulation tool that, given these order and shortcuts, automatically builds the corresponding overlay and gives some performance indicators about it.

Outline. Section 2 details the formal minimal definition of a ordered network. Section 3 discusses the linearization process of any topology into a ring, and compares by simulation, several of the most commonly encountered topologies. Section 4 summarizes the paper and gives hints on future work.

Preliminaries. Programs written in the paper use a light Java style where "**new** $Foo(\ldots field_i = E_i \ldots)$" creates an object of type Foo which implicitly has a field $field_i$ and this field, for the created object, will have the value computed by E_i. Implicitly, variables of Type Foo are named foo. Methods of Foo are declared as "$foo.method(\ldots, arg_i, \ldots) \triangleq \ldots$" where foo may be used as the classical *this* in the body of the method. Methods can be overridden. Field access and method invocation are used with the classical dotted notation ($obj.field$ and $obj.method(\ldots)$).

2 Formal Specification of Ordered Networks

A *network* is a graph of *nodes* where edges represent (physical or logical) connections. We denote by W the set of nodes of a network. The main role of the network is to dispatch messages between nodes, based on a mechanism known as the *routing* process. To locate the destination of a message, each node has a unique *identifier*, *e.g.,* IP addresses are the identifiers for the Internet. In the considered logical networks, each node is also responsible for the management of a set of identifiers (also known as *keys*), *i.e.*, a specific portion of the identifiers space. Thus, such a network does not resolve only queries such as *"send a message to the node whose identifier is* id*"*, but supports more general queries such as *"send a message to the node managing the identifier (or key)* id*"*. Note that, when nodes manage their own identifier, the second style of queries encompasses the first one. In the networks studied in this paper, we will force, by construction, the uniqueness of identifiers : an identifier is managed by one and only one node. We denote by T the domain of identifiers, and by $n.id$ the particular identifier used to identify node n. One important property to be tackled is that networks are dynamic: nodes can join and leave the network at anytime. Moreover, we are attached to guarantee that messages reach their destinations, *i.e.* that the network does not lose messages during the routing process.

2.1 Orders and Alignments

The approach we have followed is to consider a hamiltonian cycle over the network : a cycle which traverses all nodes of the graph once and only once.

Ordered nodes. Hamiltonian cycles are generally extracted from an existing graph, which is known to be an NP-complete problem. In our case, the cycle is established at the beginning, when the network is initiated with a first node, and preserved during insertion and deletion of nodes. As we consider oriented graphs, each node is only required to maintain its *successor* in the cycle. If n is a node of the network, we will denote by $n.succ$ the identifier of the successor of n in the cycle, *i.e.,* nodes are created with something resembling: "**new** $Node(id = \dots, succ = \dots)$". It is particularly convenient to consider a strict order $<$ over T. Given this order, a finite set of nodes can be sorted, thus giving a Hamiltonian path. The cycle is achieved by artificially connecting the two bounds of the path. To ensure the uniqueness of identifiers, we also assume an equality relation ($=$) on T. We have abstracted both relations $<$ and $=$ (and implicitly $\leq$) with a unique function $((T, T, bool) \to bool)$, denoted $x <_b y$, which considers the case of equality of x and y via the boolean b. If $<$ and $=$ are known, we can define this function by :

$$x <_b y \overset{\triangle}{=} x < y \ \vee \ (b \ \wedge \ (x = y)) \tag{1}$$

The strict order properties of $<$ and the equivalence properties of $=$ are reported on $<_?$ with the following definition :

$$<_? \ \textit{is a conditional order} \ \overset{\triangle}{\Leftrightarrow} \begin{cases} (reflexivity) & x <_b x = b \\ (symetry) & x <_b y = \overline{y <_{\overline{b}} x} \\ (transitivity) & x <_b y \ \wedge \ y <_c z \Rightarrow x <_{b \wedge c} z \end{cases}$$

It is easy to check that Definition 1 introduces a conditional order. Reciprocally, conditional orders are strong enough to define an order with $x \leq y \overset{\triangle}{=} x <_{true} y$, a strict order with

$x < y \triangleq x <_{false} y$ and an equivalence with $x = y \triangleq x <_{true} y \ \wedge\ y <_{true} x$.

Alignments. Considering the hamiltonian cycle, the main property we will consider is the alignment of three nodes a, b and c. Informally, these three nodes are said to be aligned if, starting from a and following the cycle, we reach b before c. As for conditional orders, we have to consider the possibility of equalities between these three nodes. Using $<_?$, this predicate can be expressed by a function $((T, T, T, bool, bool, bool) \to bool)$ defined by :

$$align(a,b,c,ab,bc,ca) \triangleq \begin{cases} \textbf{match } a <_{ab} b, b <_{bc} c \textbf{ with} \\ \mid true, true \quad \Rightarrow true \\ \mid false, false \quad \Rightarrow false \\ \mid \_, \_ \quad \Rightarrow c <_{ca} a \end{cases}$$

The two first booleans ab and bc tell if the interval is open or closed, for example $align(a, b, c, true, false, \_)$ has a meaning similar to $b \in [a, c[$. An ambiguity can appear when a and c are equal, as the interval $[a, a[$ can be either considered as the empty set or as the whole set T. The role of the last boolean ca is to give a meaning to the alignment in these particular cases. To consider $[a, a[$ as the empty set, ac should be $false$, to consider it as T, ac should be $true$. Hence, $align(a, x, a, true, false, true)$ is $true$ for all a and x. We now devise four main properties of $align$:
(i) The three nodes can be freely rolled :

$$align(a,b,c,ab,bc,ca) = align(b,c,a,bc,ca,ab)$$

(ii) The reflexivity of the conditional order implies a second property, which generalizes the previous discussion about the equality of bounds :

$$align(a,b,a,ab,\overline{ab},aa) = aa$$

(iii) Following the symetry property of conditional order, the third property shows the consequence of exchanging two nodes :

$$align(a,b,c,ab,bc,ca) = \overline{align(a,c,b,\overline{ca},\overline{bc},\overline{ab})}$$

(iv) Finally, the last main property is a consequence of transitivity :

$$align(a,b,c,ab,bc,ca) \ \wedge\ align(a,c,d,\overline{ca},cd,da) \Rightarrow align(a,b,d,ab,bc \ \wedge\ cd,da)$$

Topologic invariants. With the $align$ definition, we can formalize the invariant needed for structuring the network: (i) each node has a unique identifier, (ii) each node has a successor belonging to the network, (iii) a node manages identifiers (keys) between itself and its successor[1] and each identifier is managed by one and only one node :

$$W \ is\ well\ formed \overset{\triangle}{\Leftrightarrow} \begin{cases} (uniqueness) & \forall a, b \in W;\ a.id = b.id \Rightarrow a = b \\ (successor) & \forall a \in W;\ \exists b \in W;\ a.succ = b.id \\ (keys) & \forall k \in T;\ \exists! a \in W; \\ & align(a.id, k, a.succ, true, false, true) \end{cases}$$

The last property can be reformulated by :

$$\forall a, b \in W; align(a.id, a.succ, b, true, true, true)$$

[1] Such a simple consistent hashing scheme is for instance used in the Chord protocol, but others have been proposed in literature.

2.2 Routing

All messages received by a node are stored in a dedicated queue on this node. The main job of a node is to extract and analyze messages from this queue, one message at a time. An extracted message may either have reached its destination or must be forwarded to another node. This can be modeled by the following pseudo-code of the *analyze* method :

$$node.analyze(message) \stackrel{\triangle}{=} \left\{ \begin{array}{ll} \textbf{if} & message.arrived(node.id, node.succ) \\ \textbf{then} & message.eval(node) \\ \textbf{else} & node.forward(message) \end{array} \right.$$

where the *arrived* method tests if the message whose target is, let's say, the key k, has reached its destination :

$$message.arrived(a, b) \stackrel{\triangle}{=} align(a, k, b, true, false, true) \tag{2}$$

Therefore, by the third (*keys*) invariant, the *eval* method (evaluating the content of a message) will be called on the correct node. We will see later that specific messages use a different *arrived* method.

If a message is in transit on a node, it must be forwarded to another node. The simplest way to do so is to send this message to the successor of the node. In this way the message will traverse the cycle until reaching its destination. However, to avoid a long traversal (linear in the number of nodes), we give a chance to this message to find a shortcut :

$$node.forward(message) \stackrel{\triangle}{=} \left\{ \begin{array}{ll} \textbf{let} & link = node.shortcut(message) \ \textbf{in} \\ & link.send(message) \end{array} \right. \tag{3}$$

The *shortcut* method is a collaboration between the node and the message. The node proposes to the message all nodes it knows of[2] and the message selects those that are *shortcuts* in its own point of view. Finally, the node returns the *furthest* one (the locally best shortcut) :

$$node.shorcut(msg) \stackrel{\triangle}{=} \left\{ \begin{array}{lll} \multicolumn{3}{l}{result = node.succLink} \\ \textbf{foreach} & \multicolumn{2}{l}{link \ \textbf{in} \ node.knownLinks() \ \textbf{do}} \\ & \textbf{if} & msg.isShortcut(result.id, link.id) \\ & \textbf{then} & result = link \end{array} \right.$$

To allow for such shortcuts, we need to define the links properly. A *link* is the association of (i) an identifier, (ii) a *contact*, which is the minimal and transmissible (serializable) information required for opening a connection (*e.g.,* an IP address and a port number), and (iii) a *connection*, *i.e.,* all information needed to communicate with this node (*e.g.,* sockets). Thus a link is created with :

$$\textbf{new} \ Link(id = \ldots, contact = \ldots, \ connection = \textbf{null})$$

Note that the communication, and so the use of the *connection* field, is abstracted by the *link.send(message)* invocation as well as in the *forward* method definition. *node.succLink* is the link a node maintains to its successor (*node.succ* is equivalent to *node.succLink.id.*) A link is initially created without the physical connection which will be started by the contact. At this stage, we do not specify when this physical connection is established – an *eager* version will do

[2] In practice, nodes commonly store information about distant nodes using one or several tables.

it when the link is created, a *lazy* version will wait for the first invocation of a *send* to create this physical connection.

Quite similarly to Definition 2 of *message.arrived*, for a message whose purpose is to contact the node managing the identifier k, the *isShortcut* method should be defined as:

$$message.isShortcut(a, b) \triangleq align(a, b, k, false, true, false) \tag{4}$$

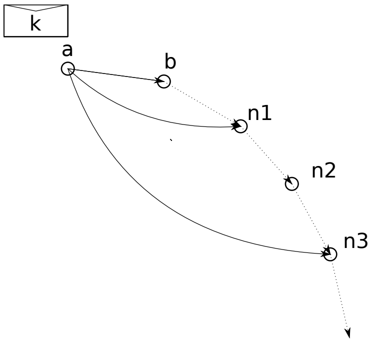

Figure 1: One routing step.

Figure 1 illustrates one routing step when a node a receives a message for the identifier k. Let us assume that n_2 is the node which manages k. Dashed lines represent the *succLink* variables. Among all links known by a, pictured with thick arrows, the furthest one but still before the destination k is chosen. Thus, for the example, Node n_1 is returned by the invocation of the *shortcut* method. The message will be sent to n_1 and, if no insertion/deletion occurs around n_1, this message will reach its destination after its transit in n_1. (n_2 is the successor of n_1.)

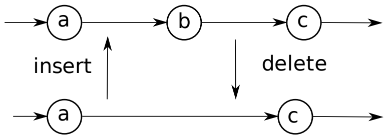

Figure 2: Insertion and deletion.

2.3 Insertion

The routing process relies on invariants to ensure that all destinations are reachable. Pursuing our formalization effort, we now have to ensure that these invariants are preserved by insertion/deletion of nodes. Figure 2 shows the informal behavior of insertion/deletion. To preserve the invariant, when inserting b between a and c, we must ensure $align(a.id, b.id, c.id, false, false, true)$. The two $false$ values ensure the uniqueness of identifiers.

For the insertion of node (known by its contact information) with a given identifier, a particular insertion message should be created with :

$$\textbf{new } \mathit{Insert}(\mathit{id} = \ldots, \mathit{contact} = \ldots)$$

On receipt, the evaluation of this message can be described as follows :

$$\mathit{insert.eval}(\mathit{node}) \triangleq \left\{ \begin{array}{l} \textbf{if } \ \mathit{align}(\mathit{node.id}, \mathit{insert.id}, \mathit{node.succ}, false, false, true) \textbf{ then} \\ \quad \textbf{let } \mathit{link} = \mathit{node.succLink} \textbf{ in} \\ \qquad \mathit{node.succLink} = \\ \qquad\qquad \textbf{new } \mathit{Link}(\mathit{id} = \mathit{insert.id}, \mathit{contact} = \mathit{insert.contact}) \\ \qquad \mathit{node.succLink.send}(\\ \qquad\qquad \textbf{new } \mathit{Start}(\quad \mathit{id} = \mathit{insert.id}, \\ \qquad\qquad\qquad\qquad \mathit{succ} = \mathit{link.id}, \\ \qquad\qquad\qquad\qquad \mathit{contact} = \mathit{link.contact})) \end{array} \right.$$

Following the definition of the method *analyze* on nodes discussed above, when an *eval* method is called on a message, this message has reached its destination. The only difference between Definition 2 of *message.arrived* and the precondition to insert a node, is the first boolean of *align*, which is set to $false$. With this restriction, we prevent a node with an already used identifier to be inserted, thus ensuring the uniqueness of identifiers in the network.

Once the invariant is ensured, the evaluation of an *insert* message consists in: (i) creating a new link with the given contact, thus initiating a connection between the creator node and the new node, and (ii) sending a message to the new node with the required information, *i.e.,* its identifier and the identifier and contact to its successor.

The *eval* method of a *Start* message consists only in setting the identifier and the link to the successor :

$$\mathit{start.eval}(\mathit{node}) \triangleq \left\{ \begin{array}{l} \mathit{node.id} = \mathit{start.id} \\ \mathit{node.succLink} = \mathit{newLink}(\mathit{start.succ}, \mathit{start.contact}) \end{array} \right.$$

Figure 3, depict the insertion steps. When b is inserted between a and c, the insertion proceeds in two steps. First, on receipt of the *insert* message, the link between a and b is created and the *start* message is sent to b. Second, on receipt of this *start* message, the link between b and c is established. Note that, between these two instants, the invariant is not clearly preserved.

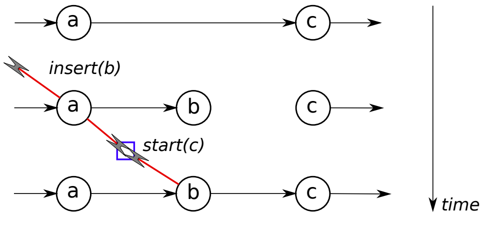

Figure 3: Insertion steps.

From a's point of view, b is active (*i.e.,* considered in W) as soon as the *insert* method is completed. After that, a can start to forward some messages to b. From b's point of view, it becomes active only on receipt of its *start* message. This hardly shows that we assume that the order in which messages are sent must be preserved on the receiver's side : if a message overtakes the *start* message during the connection between a and b, this message will not read the valid values of identifier and successor of b, leading to potential incorrect routing.

2.4 Deletion

Compared to insertion, even if at a first glance only one link needs to be updated, as shown in Figure 2, deleting a node is far more complicated. As we devise in details in the following, a node can not simply leave the network without potentially inducing inconsistencies in the routing process and losing messages.

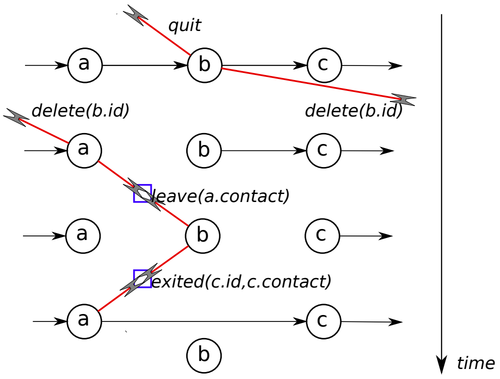

Figure 4: Deletion steps.

Roughly speaking, as illustrated in Figure 4, four steps are required for a proper deletion. Firstly, when node b decides to leave the network (we assume it does so on receipt of a *quit* message from its application layer), b first requests the collaboration of its predecessor (a on Figure 4) by initiating a particular message $delete(b.id)$. Secondly, after the routing process, when a receives this message, it sends a last message $leave(a.contact)$ to b and closes the connection with b. This last message contains the information to call back a when necessary. Thirdly, on receipt of the $leave(contact)$ message, b flushes all its pending messages, closes the connection to its successor and calls its predecessor back with the identifier and contact of c. Finally, on receipt of the $exited(id, contact)$ message, a ends the deletion steps by setting its successor with the message's arguments.

Deletion's flow. Let us now review in detail the message exchanges involved in a node's deletion. As mentioned above, the deletion process starts when a node b receives a *quit* message from its application layer. Since only the predecessor of b can perform the deletion, this *quit* message is transformed into a $delete(b.id)$ message whose informal meaning is : *if your successor is identified by b.id, then delete it.* It would clearly be an improvement for each node to have a direct access to its predecessor, so it could contact it directly on deletion. However, if one looks back to Figure 3, during insertion, the predecessor of a node is not always clearly defined. This requires for the *delete* message to follow the general routing process :

$$quit.eval(node) \triangleq node.analyze(\mathbf{new}\ Delete(id = node.id))$$

A *delete* message does not implement the general *arrived* and *isShortcut* methods described in Definitions 2 and 4. It traverses the ring until it reaches a node whose successor's identifier is the node to be deleted :

$$delete.arrived(a, b) \triangleq (b = delete.id)$$

$$delete.isShortcut(a, b) \triangleq align(a, b, delete.id, false, false, false)$$

$$delete.eval(node) \triangleq node.succLink.send(\mathbf{new}\ Leave(contact = node.contact))$$

If the notion of predecessor (*e.g.,* $x-1$ for natural numbers) is defined for Type T, we can keep the standard definition of *arrived* and *isShortcut* by sending a message $delete(pred(node.id))$ to reach the predecessor. However, not every type T necessarily supports a *pred* operator. This is the case, for instance, of rational numbers and strings of arbitrary length (*e.g.,* the predecessor of "foo" is "fonzzzz. . . " with an undefined number of 'z').

As expected, a node evaluating a *Delete* message transmits a *Leave* message to its successor and closes the connection. Note that, since this message will be the last one sent with this connection, some improvements are possible in the *close* protocol (for example, some steps for closing a TCP connection can be removed). Even if the essential meaning of a *Leave* method is to reply with the current successor to the sender, the main problem is then to ensure that the leaving node will not receive further messages. As mentioned in the *eval* method of *Delete* messages, the predecessor closes its link, but all others links to the leaving node, open for shortcut purposes, are still alive. The first action a node has to do on a *Leave* arrival is to close all incoming links. We assume that a *closeIncomings*() method is provided for this. At this stage, if the node refuses all new connection, the queue of incoming messages may be non-empty, but cannot grow anymore. Finally, to exit, the leaving node can send to itself a specific *Shutdown* message :

$$leave.eval(node) \triangleq \begin{cases} node.closeIncomings() \\ node.send(\mathbf{new}\ Shutdown(contact = leave.contact)); \end{cases}$$

Due to the previous *closeIncomings*(), a *Shutdown* message is necessarily the last message to be received on a leaving node. In other words, when a node evaluates a *Shutdown* message, its queue of incoming messages is empty. The only thing that remains to be done then is advising its predecessor, with an *Exited* message, giving the necessary information to set its new *succLink* :

$$shutdown.eval(node) \triangleq \left\{ \begin{array}{ll} \mathbf{let} & link = shutdown.contact.open()\ \ \mathbf{in} \\ & link.send(\\ & \quad \mathbf{new}\ Exited(id = node.succ, \\ & \qquad\qquad contact = node.succLink.contact)) \\ & node.succLink.close() \end{array} \right.$$

The deletion process ends when the predecessor evaluates the *Exited* message whose purpose is only to set the link to its new successor :

$$exited.eval(node) \triangleq node.succLink = \mathbf{new}\ Link(id = exited.id, contact = exited.contact))$$

Deletion automata. The deletion process shown on Figure 4 is correct only if no other messages are sent to the leaving node concurrently. For instance, let us assume node a receives an *insert* message after sending the *leave* message, then a must clearly wait for the receipt of the *exited* message before performing the insertion, otherwise it is unable to provide the correct successor information to the inserting node. To be more accurate, let us provide a *state* to each node of the network. The state of a node changes depending of the message it receives. Figure 5 shows the state automaton of nodes. Vertices represent the possible states of nodes, and edges are the possible transitions between states. Edges are labelled with $\mathbf{m_1}(\mathit{code}) \to m_2$ representing the fact that, on transition, the node receives the message m_1, executes *code* and sends the message m_2. At the beginning of its lifetime, *i.e.,* on receipt of the first *start* message, a node's state is set to *Run*. It remains so until receiving either a *quit* (from its application layer) or a *delete* message (from its successor).

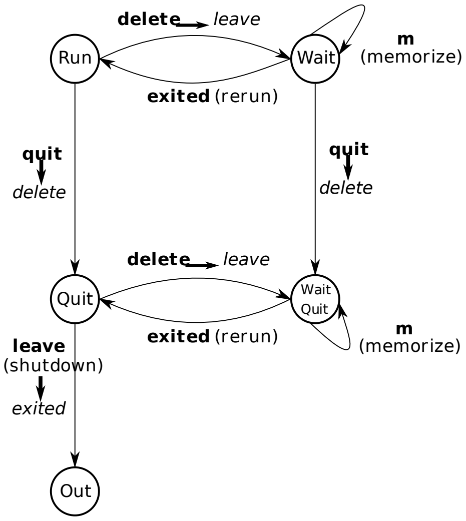

Figure 5: Node state automaton.

In the *Quit* state, a node waits for the *leave* message but may itself be requested to delete its successor (via the reception of the dedicated *delete* message.) *Wait* and *WaitQuit* states are similar except that in the *Wait* state, a node is deleting its successor, but may be requested to leave (by the reception of a *quit* message from its application layer). (Note that this may occur only once in a node's lifetime). If a is in *Wait* state, it waits for the actual termination of its old successor b. As long as the new successor c is not established, messages involving the old successor b, *i.e.,* the node to be deleted, must be memorized. Once the *exited* message is received, all memorized messages can be pushed back in a's queue for future analysis.

This memorization also concerns *insert* messages. An *insert* message can not be processed in this state since we do not have a properly established successor to send to the inserted node in its *start* message. Nevertheless, all other messages can be either evaluated if they do not concern insertion, or forwarded to their next hop, as far as b is not the target. In the same way, in a *WaitQuit* state, an incoming *leave* message will be memorized until the node goes back to the *Quit* state, as a node has to establish its new successor before leaving the network (*delete before leaving*). In a *WaitQuit* state, an *exited* message sent after receiving a *leave* message would contain information on the successor which is not valid.

Finally, note that the memorized messages can not be directly forwarded to the new successor c when the *exited* acknowledgment is received: some of these messages may concern a itself, since, by receiving the *exited* acknowledgment, a now manages the identifiers of the deleted node b.

Chains of deletions. If some of successive nodes decide to leave the network at the same time, it may appear a chain of nodes all with the state $WaitQuit$, all waiting for the *exited* acknowledgment of their successor. Only the rightmost node can start sending the acknowledgment. In other words, the chain is destroyed by the right.

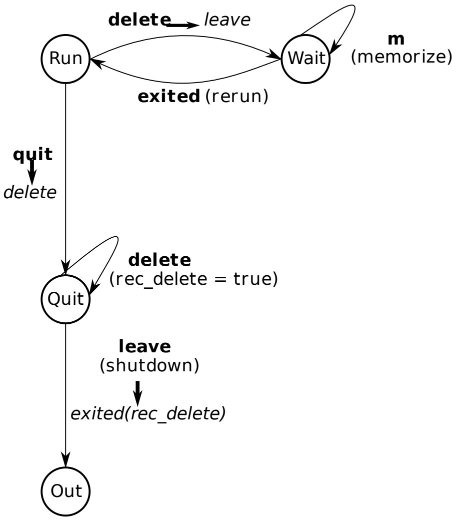

Figure 6: Node state automaton (left).

Figure 6 shows another automaton where chains of leaving nodes are destroyed by the left. This automaton is a simplification of the one pictured in Figure 5. Similarly, a *quit* message is memorized in a $Wait$ state, and a *Quit* state prevents a *leave* message to be emitted on receipt of a *delete* message by memorizing it. Note that, with this new automaton, a node in a *Quit* state can receive only one *delete* message. A boolean is sent in the *exited* message indicating if a *delete* message was received. Thus, when a node in $Wait$ state receives an $exited(true)$ message, it has to stay alive to delete its new successor. In contrary, on receipt of an $exited(false)$ message, a node can freely return to its *Run* state, having the opportunity to (re)evaluate a *quit* message memorized during its $Wait$ state.

Solving the deletion deadlock. Unfortunately, both automata in Figures 5 and 6 contain deadlocks. In the first one, it appears when all nodes of network reach the $WaitQuit$ state at the same time.

Figure 7 shows a simple sequence of messages leading to a deadlock in a network of two nodes a and b following the automaton depicted by Figure 5. At approximately the same time, nodes a

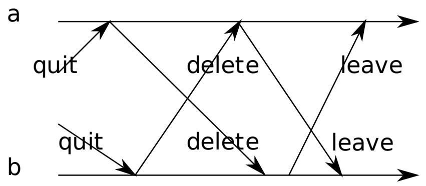

Figure 7: Deadlock in a network of two nodes.

and b decide to leave the network. So both of them emit their *delete* message and move to a *Quit* state. If they receive this *delete* at the same time, they both emit a *leave* and move to $WaitQuit$ state. But those *leave* will be memorized and never analyzed. For the second automaton, the deadlock appears more quickly since the *delete* messages will not be analyzed.

It is easy to check that in a network of two nodes a and b, if a behaves according to one automaton, and b according to the other, then no deadlock can occur. The property remains true for any network, as long as it exists at least one node that acts according to one automaton and one node according to the other. This constraint can be easily preserved on insertion/deletion if exactly one node, named the *leader*, acts according to one of the two automata (all the other nodes acting according to the other one). A network with only one node must contain the leader.

There are two ways to determine the leadership. The first way is to elect as leader the node which satisfies the property $node.id \geq node.succ$. This node is unique: it is the node with the highest identifier with respect to the order. In this case, the leadership may change on insertion/deletion: (1) If the leader inserts a node with a higher identifier, it releases its leadership to the new node. (2) Any node deleting the leader becomes the new leader. But, looking closer to this second rule, we can check that it is sufficient to ensure the uniqueness of the leader: a leader can remain the leader until it is deleted, and the node deleting it becomes the new leader. The only drawback of this second method is that another boolean must be sent in *exited* message stating if the deleted node was the leader or not.

3 Concretization of Ordered Networks

As our specification and algorithms are generic and provable for an oriented ring overlay network, it gives the opportunity to formally study any topology that can be *mapped* onto an oriented ring, or as we already mentioned, that contains a Hamiltonian cycle, which appears to be the case for many topologies commonly encountered in P2P systems (cartesian spaces, hypercubes, butterflies, trees, etc.) To apply our specification to a given topology T, we need to consider two things, namely *mapping* and *shortcut.*

Mapping. The first thing needed is to find a mapping of T onto an oriented ring, *i.e.,* determine a function associating a node to a unique *id* reflecting its location on the hamiltonian cycle. In other words, we have to find an order on the nodes. Once the topology is mapped, the previously presented algorithms can be apply, except those related to the order itself.

Shortcuts. As we specified in Section 2, each node maintains a set of routing *links*. Then, for each topology having its own properties (diameter, symmetry, ...), we need to find relevant links allowing to route queries efficiently in T. Sometimes, these links are intrinsically offered by T. Consider for instance the hypercube. Routing in the hypercube can be achieved using the neighbors of nodes (neighbors of n are all nodes having an identifier differing from $n.id$ on one and only one bit.)

Landmarks. However, we need to take into account the fact that only a subset of possible nodes are effectively in the network. (The set of *ids* is larger than the set of actual nodes.) Moreover, this subset is changing as nodes are joining and leaving the network. Thus, we need to define the set of *ids* considered as *perfect* potential shortcuts. These perfect shortcuts represent the ideal configuration towards which the set of actual *links* should tend. This set is referred to as *landmarks* henceforth. When starting, a node n computes its landmarks. Then, during its lifetime, a node maintains the best link possible for each *id* in *landmarks*.

Learning. Now, we can define the *shortcut* which is the association of a landmark *id* and a *link*. Recall that a *link* is a pair $(link.id, link.contact)$ where $shortcut \triangleq (landmark, link)$. As nodes will exchange messages for routing, they can include in these messages some information about their own set of links in order to improve shortcuts of other nodes. On receipt of such a message, a node may replace the current link of one of its *shortcut* by one *link* included in the incoming message if *link.id* is closer to *shortcut.landmark* than *shortcut.link.id*, applying the following algorithm:

$$shortcut.learn(newlink) \triangleq \begin{cases} \textbf{if } align(shortcut.link.id, newlink.id, \\ \qquad shortcut.landmark, false, true, false) \textbf{ then} \\ \quad shortcut.link = newlink \end{cases}$$

In the following, we discuss several orders implemented on top of our model, using a Java-based simulation tool we developed, that, given an order and a set of landmarks, is able to build the corresponding overlay and to simulate its behavior in regard to some key performance indicators (mainly the routing complexity). The first order, *ring*, introduces some general notions and notations that may be reused in the following orders. For all models, N denotes the number of nodes.

3.1 Ring

Natural numbers are obvious candidates to order things. Our first topology is based on positive integers with the natural order based on them. The landmarks maintained for routing purpose are directly inspired by the Chord approach, *i.e.,* exponentially distant of the considered node [21]. More precisely, a node n has a set of landmarks with identifiers $p.id + 2^i$ for $i > 0$ (the case $i = 0$ is redundant with *succLink*). The only remaining question is: how many shortcuts a node is supposed to maintain? If, at the application layer, inserted nodes have bounded identifiers, say for example that all identifiers fit in a 64-bit integer, then all nodes have to reserve space for 64 shortcuts. Otherwise, during its learning process, a node can memorize the *highest* identifier encountered. If the approximation of highest identifier is kept in the node's local variable max, the number of shortcuts must follow dynamically the value $\lceil log_2(max) \rceil$.

A natural measure of the efficiency of networks is the average of node a message must traverse before reaching its final destination. This number is correlated to the latency of the network, and is commonly referred to as the *diameter of the network*, or the *number of hops*. The distribution of nodes' identifiers and the insertion/deletion of nodes influence it deeply. Nevertheless, it makes sense to compute a theoretical value when the network is saturated ($W = T$ and $card(T) = 2^k$) and stable (no insertion/deletion). We will note μ this theoretical value, which depends only on N. For networks based on natural numbers, the natural order and the shortcuts presented above, it is easy to verify that $\mu = \frac{log_2(N)}{2}$.

3.2 Hypercube

Another topology explored to build overlay networks is the hypercube, or more generally the n-cube [19]. As previously mentioned, identifiers of nodes in a hypercube are binary identifiers, and the neighbors of a node n are every node having an identifier differing by only one bit to the identifier of $n.id$. These identifiers are used as landmarks for the node. As a consequence, natural candidates to find a hamiltonian cycle in a hypercube are the Gray codes [6]. Similarly, a gray code is a binary numeral system where two successive values differ by only one bit. The gray code can be recursively obtained through the following definition (*n-ary Gray Code*):

$$A_i = \{0 \bullet A_{i-1}, 1 \bullet {A_{i-1}}^{-1}\}$$

where A is a sequence of binary strings, A^{-1} denotes the same sequence in reverse order, and $a \bullet A$ concatenates the bit a to every element of the sequence A. Following this definition, the initial sequence $A_1 = 0, 1$ is refined into $A_2 = 00, 01, 11, 10$, and then into $A_3 = 000, 001, 011, 010, 110, 111, 101, 100$ at step 3. The mapping obtained through the gray code is illustrated on Figure 8.

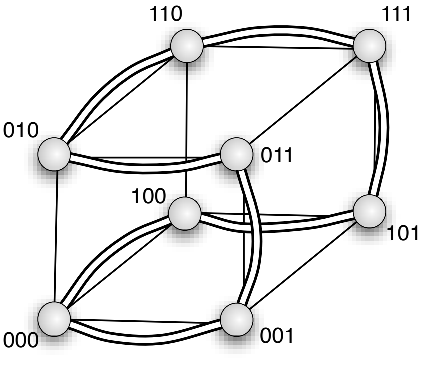

Figure 8: Linearizing a 3-cube.

Following the gray code generation previously given, we need to write the algorithm ordering two identifiers. Let $a.x$ denotes the concatenation of the bit a to the strings representing the value x, ε is the empty string, $\|x\|$ the size of the strings representing the value x, moreover, before calling the order predicate, we assume that all the leading zeros of both arguments are removed. As a consequence, all values, except zero, start with a "1". This leads to the following recursive algorithm :

$$x <_b y = \begin{cases} \|x\| < \|y\| & \text{if } \|x\| \neq \|y\| \\ b & \text{if } x = \varepsilon \\ x' <_b y' & \text{if } x = 0.x' \wedge y = 0.y' \\ True & \text{if } x = 0.x' \wedge y = 1.y' \\ False & \text{if } x = 1.x' \wedge y = 0.y' \\ y' <_b x' & \text{if } x = 1.x' \wedge y = 1.y' \end{cases}$$

Note that, starting from hypercube and Gray code originally defined on a set of 2^k elements, we end up here with a general definition of a conditional order using bit-string of *arbitrary* length. The n-cube intrinsically allows logarithmic routing by using neighbors of nodes, at each step decrementing the number of differing bits between current node and destination. The previous discussion about dynamically setting the number of shortcuts in a ring topology also holds for n-cubes.

For this topology we have computed the theoretical latency which is : $\mu = \frac{3}{4}(log_2(N)-1)+\frac{1}{N}$.

3.3 Cartesian Space

Another topology commonly used by P2P systems is the cartesian space, such as in the early CAN approach [17]. Moreover, cartesian spaces are good candidates to support more complex queries (such as range queries) than the usual lookup for a fixed *key*. Note that an interesting formalization of CAN through π-calculus can be found in [9].

Linearizing the space. In a cartesian space, each node is a point determined by a set of coordinates. As illustrated by Figure 9, ordering points can be obtained through space filling curves (SFCs) [1] which give a continuous mapping from a d-dimensional space to a 1-dimensional space. Imagine for instance a d-dimensional cube with the SFC passing through each point in the cube volume, and entering and exiting the cube only once. Given a point of a d-dimensional space, the SFC returns a real value between 0 and 1, while preserving the locality, *i.e., close* points in the d-dimensional space will obtain *close* values when projected on the unit vector. Among SFCs, the Hilbert SFC has been proven to be the one that preserves proximity of nodes the most [1].

The mapping process is illustrated by Figure 9, Each node, by its coordinates, *owns* one small square. The 1-dimensional indexes are obtained recursively, by refining the curve. The index is then the number of squares traversed starting from square zero (bottom left).

Each refinemenet step follows a strict scheme. The curve is made of a succession of four patterns, each one being refined in a precise way. The refinement of the four patterns are illustrated by Figure 10. One way to perceive the refinement step is the following. For clarity, we number them 0, 1, 2, and 3. For instance, one refinement step of Pattern 0 produces a new partial curve being the concatenation of patterns 1, 0, 0, 2, in this order. As illustrated by Figure 10, the other refinements are very similar : only a rotation is needed.

Following these patterns and their sub-patterns, we can design an order on points. Let $A = (x_A, y_A)$ and $B = (x_B, y_B)$ two points whose both coordinates have the same number of bits. If necessary, the shortest of the two can be left-padded with zeros. x_A denotes A's abscissa and y_A denotes its ordinate. Let x_a (resp. y_a) be the leftmost bit of x_A (resp y_A).

Let us take the convention, adopted on Figure 9, that the first refinement (creating a square of size 2) follows pattern 0. As pictured in Figure 10, the first pattern in the refinement of

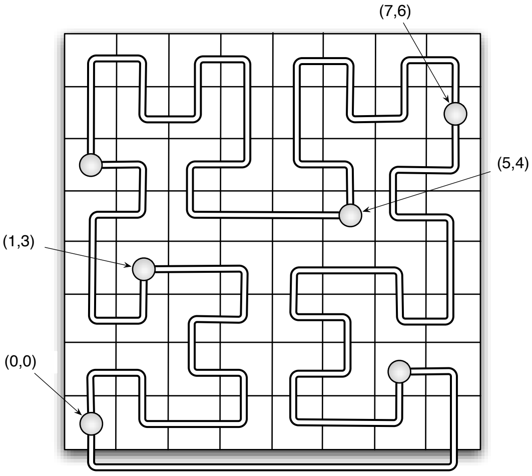

Figure 9: Linearizing a tore.

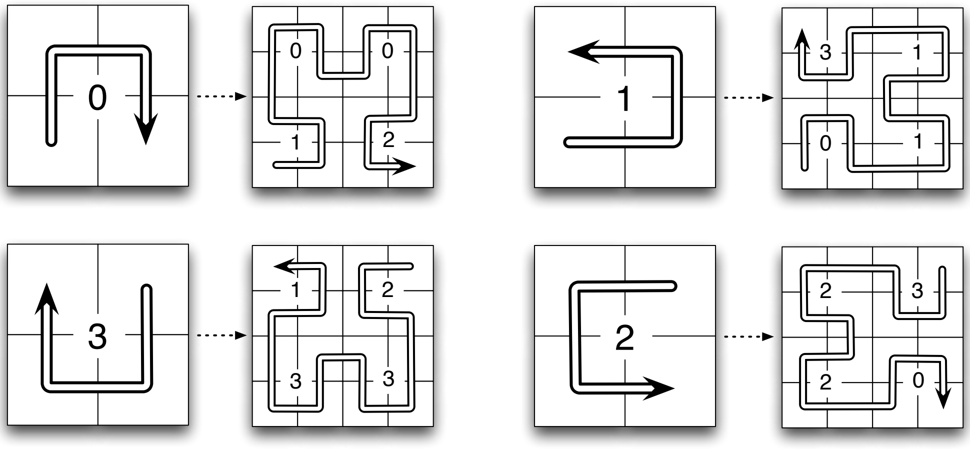

Figure 10: One Hilbert refinement step.

pattern 0 is pattern 1, and the first pattern in the refinement of pattern 1 is pattern 0. This means that the outmost pattern is always either 0 or 1, depending on the parity of the number of refinements. In short, if $\|x_A\| = d$, the outmost pattern is $(d+1)\&1$. Then, to compute $A <_b B$, we call an auxiliary predicate $A <_b^p B$ where p denotes the current pattern. On first call, $p = (\|x_A\|+1)\&1$. Let us denote A' and B' such that $x_A = x_a.x'_A$, $x_B = x_b.x'_B$, $y_A = y_a.y'_A$ and $y_B = y_b.y'_B$. Then :

$$A <_b^p B = \begin{cases} b & \text{if } A = \varepsilon \\ A' <_b^{nextPat(p,x_a,y_a)} B' & \text{if } x_a = x_b \wedge y_a = y_b \\ resPat(p, x_a, y_a, x_b, y_b) & \text{otherwise} \end{cases}$$

The first case is trivial ($\varepsilon <_b^p \varepsilon = b$). Otherwise, if A and B are in the same sub-square (second line), we cannot give an answer now and we have to make a recursion. We need then to determine the next pattern. This is obtained through the *nextPat* function whose parameters are the current pattern and the leftmost bits of coordinates of A and B. This function can be easily written looking at Figure 10. For instance, if we are in the pattern 2 and $x_a = y_a = 1$, then A and B are in the up-right sub-square and $nextPat(2, 1, 1)$ returns 3. A' (resp. B') is constructed by removing the leftmost bit of the coordinates of A (resp. B). These new coordinates are the one relative to the sub-square.

Finally, the third case is when A and B do not belong to the same sub-square (either $x_a \neq x_b$ or $y_a \neq y_b$). In this case, the answer can be computed with the current pattern and these bits, through the *resPat* function. For example, if the current pattern is 2, A is in the up-right sub-square ($x_a = y_a = 1$) and B is in the down-left sub-square ($x_b = y_b = 0$), then A is before B in the linearization: $resPat(2, 1, 1, 0, 0) = true$. Once again, the values returned by *resPat* can be extracted by a simple reading of Figure 10. Note that, as for the ring and hypercube topologies, we have defined an order independent of the sizes of coordinates.

Landmarks in the space. In CAN, each node is responsible for a square portion of the space, and maintains a link to all nodes responsible for a square contiguous to its own square. This routing scheme was shown to be efficient (with a logarithmic complexity) only if the number of dimension of the space is fixed to $\log(N)$. However, this requires to be able to dynamically determine the number of nodes in the network, and reset the number of dimensions when nodes are joining or leaving the network, which can be very costly at large scale. Instead, we will here try to use the nature of cartesian space to determine good routing links for the basic topology, *i.e.,* when $d = 2$, for which case CAN's routing is not logarithmic.

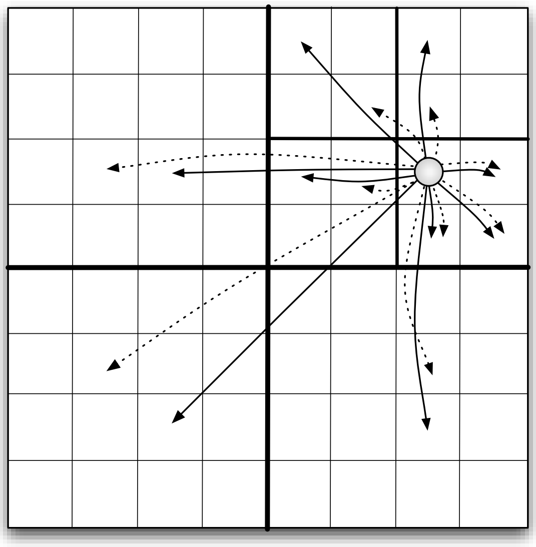

Figure 11: Routing links in a 2-dimensional space.

- We can first choose to change only some of the bits x_i and y_i, for example, for the landmark in the same column, we can take $(X, Y_h \bullet \overline{y_i} \bullet Y_l)$. In this case, the landmarks have the same position as A, but inside their respective sub-squares. This set of links is pictured on Figure 11 with solid arrows.

- Instead of leaving X_l and Y_l as it is, we can reverse all these bits, then for the landmarks in the same column we have $(X, Y_h \bullet \overline{y_i} \bullet \overline{Y_l})$. In this case the landmarks take place at some symmetric position relatively to the horizontal median, the vertical median, or the central point. This second possibility is pictured on Figure 11 with dashed arrows.

- At each level, we can also choose a random point in the sub-square.

3.4 Simulation results

To compare the three topologies described above we have chosen to analyse stable and well distributed networks, *i.e.,* where no insertion/deletion of nodes occurs and when each node manages the same number of identifiers. Note that our goal here, is not compare the performance of the topologies we describe before, as they already have been well-studied, but more to present the results of the use of our java-based tool whose goal is to construct and simulate overlays based on an ordering function and a set of landmarks.

In the following, if N is not a power of two, the number of managed identifiers may differ by one between two nodes. We have chosen a domain of identifiers with a size of 4K, *i.e.,* , integers on 12 bits for Ring and Hypercube, and a Cartesian Space with coordinates on 6 bits. For each number of nodes N ranging from 1 to 2048, *i.e.,* until the network is half-saturated, we compute the average number of hops.

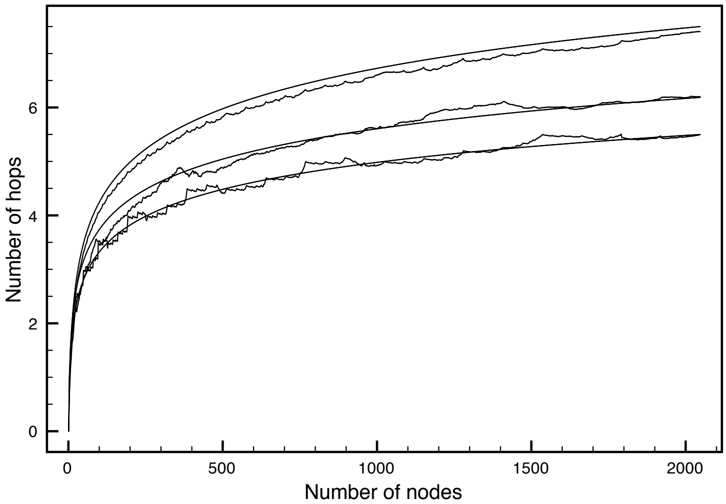

Figure 12: Number of hops Vs number of nodes.

Figure 12 shows the results for the orders presented in this section. The best one is the ring, the associated spline is $1/2 * log_2(x)$, confirming the theoretical result obtained for a saturated ring. In the middle, we find the cartesian space (here following the symmetric way of setting landmarks), which seems to follow $9/16 * log_2(x)$. The worst result is for Hypercube which is just under the $3/4 * log_2(x)$ function.

4 Conclusion

This paper studied some results that can be applied for families of networks relying only on an order on nodes' identifiers. The protocols for nodes' insertion and deletion were reviewed in details to ensure the absence of deadlocks and livelocks. All this first part of the work, presented in Section 2, was specified in Coq, a formal proof management system [3]. We have reached some strong lemmas on alignments, on topologic invariants and for the correctness of the routing process. However, we have not yet fully finished the deadlock/livelock free theorem (termination of the routing process for one message, in other words, that all messages will reach their final destination). Note that the deletion's deadlock was discovered during the attempt of solving the termination's theorem.

We have also shown, in Section 3, some concretizations of the abstract order described in Section 2. These concretizations, and thus the underlying abstract model, was implemented in Java : 750 lines for the abstract model and 750 lines for the specification of some 10 different orders. This implementation was the indispensable tool for computing the results given in Figure 12. We have also used this tool for studying different strategies for *learning*, *i.e.,* the way a node computes more accurate shortcuts when nodes join and leave dynamically.

References

[1] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-Filling Curves and their Use in the Design of Geometric Data Structures. *Theoretical Computer Science*, 181, 1997.

[2] Tarun Bansal and Neeraj Mittal. A scalable algorithm for maintaining perpetual system connectivity in dynamic distributed systems. In *IPDPS*, pages 1–12. IEEE, 2010.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[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'04*, pages 250–265, 2004.

[5] T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list. In *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2008)*, volume 5340 of *Lecture Notes in Computer Science*, pages 124–140. Springer, 2008.

[6] E. N. Gilbert. Gray Codes and Paths on the n-Cube. *Bell System Tech Journal*, 37:815–826, 1958.

[7] T. Hayes, N. Rustagi, J. Saia, and A. Trehan. The forgiving tree: a self-healing distributed data structure. In *27th Annual Symposium on Principles of Distributed Computing (PODC 2008)*, pages 203–212. ACM, 2008.

[8] T. Hayes, J. Saia, and A. Trehan. The Forgiving Graph: a Distributed Data Structure for Low Stretch under Adversarial Attack. In *28th Annual Symposium on Principles of Distributed Computing (PODC 2009)*, pages 121–130. ACM, 2009.

[9] Adrian Iftene and Gabriel Ciobanu. Formalizing peer-to-peer systems based on content addressable network. *International Journal of Computers, Communications & Control*, I(S.):268–273, June 2006.

[10] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid. A Self-stabilizing and Local Delaunay Graph Construction. In *20th International Symposium on Algorithms and Computation (ISAAC 2009)*, pages 771–780, 2009.

[11] Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In Srikanta Tirthapura and Lorenzo Alvisi, editors, *PODC*, pages 131–140. ACM, 2009.

[12] David R. Karger and Matthias Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory Comput. Syst.*, 39(6):787–804, 2006.

[13] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. Re-chord: a self-stabilizing chord overlay network. In *23rd Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, pages 235–244. ACM, 2011.

[14] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19(2):126–148, 2006.

[15] Nancy A. Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in distributed hash tables. In *First International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, volume 2429 of *Lecture Notes in Computer Science*, pages 295–305. Springer, 2002.

[16] Hyojin Park, Jinhong Yang, Juyoung Park, Shin Gak Kang, and Jun Kyun Choi. A survey on peer-to-peer overlay network schemes. In *10th International Conference on Advanced Communication Technology*, 2008.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, pages 161–172, 2001.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, pages 329–350, 2001.

[19] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. Hypercup - hypercubes, ontologies, and efficient search on peer-to-peer networks. In *AP2PC*, pages 112–124, 2002.

[20] Cristina Schmidt and Manish Parashar. Squid: Enabling Search in DHT-Based Systems. *J. Parallel Distrib. Comput.*, 68(7), 2008.

[21] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399