

Elsevier Editorial System(tm) for Theoretical Computer Science
Manuscript Draft

Manuscript Number: TCS-D-09-00039

Title: Tiara: A Self-Stabilizing Deterministic Skip List and Skip Graph

Article Type: Special Issue: Self-Stabilization (Dolev

Section/Category: A - Algorithms, automata, complexity and games

Keywords:

Corresponding Author: Mr Thomas Clouser,

Corresponding Author's Institution: Kent State University

First Author: Thomas Clouser

Order of Authors: Thomas Clouser; Mikhail Nesterenko; Christian Scheideler

Tiara: A Self-Stabilizing Deterministic Skip List and Skip Graph

Thomas Clouser^a, Mikhail Nesterenko^{*,a,1}, Christian Scheideler^b

^a*Department of Computer Science, Kent State University, Kent, OH, USA*

^b*Institute of Computer Science, Technical University of Munich, Garching, Germany*

Abstract

We present *Tiara* — a self-stabilizing peer-to-peer network maintenance algorithm. *Tiara* is truly deterministic which allows it to achieve exact performance bounds. *Tiara* allows logarithmic searches and topology updates. It is based on a novel *sparse 0-1 skip list*. We then describe its extension to a ringed structure and to a skip-graph.

Key words:

1. Introduction

Due to the rise in popularity of peer-to-peer systems, dynamic overlay networks have recently received a lot of attention. An overlay network is a logical network formed by its participants across a wired or wireless domain. The number of users of a peer-to-peer system may reach millions. Therefore, the research focuses on designing scalable and efficient overlay networks [1, 2, 3, 4, 5, 6, 7, 8, 9]. Two usual optimization parameters are the speed of searching for items in the network and the speed of topology updates.

A popular design of peer-to-peer networks is based on a sorted list or a ring [2, 3, 5, 9]. To decrease the network diameter and improve the search speed, the peers maintain a set of shortcut links to the nodes progressively further away. A skip list [10, 11] is an alternative to a balanced tree structure that is built on the basis of such list. It enables searches and topology updates

*Corresponding author

Email addresses: `tclouser@kent.edu` (Thomas Clouser), `mikhail@cs.kent.edu` (Mikhail Nesterenko), `scheidel@in.tum.de` (Christian Scheideler)

¹This research is supported in part by NSF Career award CNS-0347485.

that are proportional to the logarithm of the number of nodes in the network $|N|$. A skip graph [2, 3, 5] is an extension of a skip list is an extension of a skip graph that preserves its search and topology update properties while decreasing the network's congestion — a measure of the ability of the network to handle concurrent search requests and increasing its expansion — a measure of network's connectivity.

In open peer-to-peer systems, participants may frequently enter and leave the overlay network either voluntarily or due to failure. The scale of a peer-to-peer system dictates that faults and inconsistencies should be regarded as the norm rather than an exception. Hence, overlay networks require mechanisms that continuously counter such disturbances. Simplistic ad hoc approaches that handle individual fault conditions do not adequately perform in case of unanticipated, complex or systemic failures. In practice many peer-to-peer systems, such as KaZaA, Bittorrent, Kademlia, use heuristic methods in order to maintain their topology. Solutions presented in the research literature focus on efficiency of the proposed structure while offering only ad hoc solutions to fault tolerance.

One can argue that if nodes are randomly distributed, a sorted list or ring with a sufficient number of redundant connections will not disintegrate with high probability. However, it is not clear whether practical systems always satisfy such randomization assumption. In addition, the problem of generating high-quality trusted random numbers in a peer-to-peer systems is far from trivial and it is known that an adversary can quickly degrade the randomness of the peer-to-peer system even if perfectly random numbers are reliably generated [12]. Moreover, if a faulty state is improbable, it is not impossible. Failure to consider recovery from such states may lead to catastrophic consequences. Thus, some researchers [13, 14] argue that overlay network architects need to consider holistic approaches to fault tolerance and recovery, such as self-stabilization.

In this paper we present Tiara. It is a self-stabilizing deterministic skip-list. We extend Tiara to deterministic a skip graph. To the best of our knowledge Tiara is the first such structure with the property of self-stabilization.

Related literature. Several algorithms presented in the literature focus on stabilizing parts of overlay networks. Onus et al. [14] present several high-atomicity solutions to linearizing an overlay network. Shaker and Reeves [15] describe a distributed algorithm for forming a directed ring network topology. Héroult et al. [16] describe a spanning tree formation algorithm for overlay

networks. Cramer and Fuhrmann [17] show that ISPRP — a ring-based overlay network is, in certain cases, self-stabilizing. Caron et al. [18] describe a snap-stabilizing prefix tree for peer-to-peer systems. Bianchi et al. [19] present a stabilizing search tree for overlay networks optimized for content filters.

Several randomized overlay network algorithms have also been proposed. Dolev and Kat [20] introduce the HyperTree and use it as a basis for their self-stabilizing peer-to-peer system. Dolev et al. [21] describe a self-stabilizing intrusion-tolerant overlay network.

Our contribution. In this paper we present Tiara. It stabilizes a novel 0-1 distributed skip list. Specifically, we demonstrate a self-stabilizing algorithm for a sorted list and then show how to extend it to a self-stabilizing algorithm for a skip list. Tiara can construct these structures without any knowledge of global network parameters such as the number of nodes in the system; each node utilizes only the information available to its immediate neighbors. Moreover, Tiara preserves network connectivity so long as the initial network is connected. That is, Tiara reconstructs the connectivity of the base sorted list on the basis of skip list links. We rigorously prove Tiara correct in an asynchronous communication register based model.

We show how Tiara can be extended to form ring structures and prove the correctness of the resultant algorithm. On the basis of Tiara we develop an algorithm that maintains a deterministic skip graph. We prove that the search cost in this skip graph is logarithmic while the update cost is polylogarithmic. Note that the congestion and expansion of a skip list are in $\Theta(|N|)$ and $\Theta(1/|N|)$ respectively. We analyze the skip graph produced by our extension to Tiara and demonstrate that its congestion decreases to $O(|N|^{1/\log 3})$ and its expansion correspondingly increases to $\Omega(1/|N|^{1/\log 3})$.

Organization of the paper. First, we introduce our computational model. Then, we describe a self-stabilizing algorithm for the sorted list and formally prove it correct. We then extend it to a ringed structure and a skip graph. We complete the paper with future research directions and open problems.

2. Model

A peer-to-peer system consists of a set N of processes. Each process has a unique integer identifier. A process contains a set of variables and actions.

An action has the form $\langle name \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. $name$ is a label, $guard$ is a Boolean predicate over the variables of the process and $command$ is a sequence assigning new values to the variables of the process. For each pair of processes a and b , we define a Boolean variable (a, b) that is shared among them. Two processes a and b are *neighbors* if this variable is **true**. The *neighborhood* of a process a is defined as the set of all of its neighbors.

Sets of neighbors may be maintained on different *levels*. A neighborhood of process a at level i is denoted as $a.i.NB$. The *right neighborhood* of a , denoted $a.i.R$, is the set of neighbors of a with identifiers larger than a . That is, $a.i.R \equiv \{b : b \in a.i.NB : b > a\}$. Similarly, the *left neighborhood* of a , denoted $a.i.L$, are a 's neighbors with smaller identifiers. That is, $a.i.L \equiv \{b : b \in a.i.NB : b < a\}$. Naturally, the union of $a.i.R$ and $a.i.L$ is $a.i.NB$.

When describing a link we always state the smaller identifier first. That is, a is less than b in (a, b) . Two processes a and b are *consequent* if there is no process c whose identifier is between a and b . That is, $\mathbf{cnsq}(a, b) \equiv (\forall c :: (c < a) \vee (b < c))$. The *length* of a link (a, b) is the number of processes c such that $a < c < b$. By this definition the length of a link that connects consequent processes is zero.

A *system state* is an assignment of a value to the variables of each process. An action is *enabled* in some state if its guard is **true** at this state. A *computation* is a maximal fair sequence of states such that for each state s_i , the next state s_{i+1} is obtained by executing the command of an action that is enabled in s_i . This disallows the overlap of action execution. That is, action execution is *atomic*. The execution of a single action is a *step*. Maximality of a computation means that the computation is infinite or it terminates in a state where none of the actions are enabled. Such state is a *fixpoint*. In a computation the action execution is *weakly fair*. That is, if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often. This defines an *asynchronous* program execution model.

A state *conforms* to a predicate if this predicate is **true** in this state; otherwise the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**. Let T and U be predicates over the state of the program. Predicate T is *closed* with respect to the program actions if every state of the computation that starts in a state conforming to T also conforms to T . Predicate T *converges* to U if T and U are closed and any computation starting from a state conforming to T contains a state conforming to U . The program *stabilizes* to T if **true**

process u
variables
 $u.0.NB$ — set of neighbor processes of u .
shortcuts
 $u.0.L \equiv \{z : z \in u.0.NB : z < u\}$, $u.0.R \equiv \{z : z \in u.0.NB : z > u\}$
actions
grow right: $(s \in u.0.R) \wedge (t \in s.0.L) \wedge (t \notin u.0.NB) \longrightarrow$
 $u.0.NB := u.0.NB \cup \{t\}$
trim right: $(s, t \in u.0.R) \wedge (t \in s.0.L) \wedge (\forall z : z \in u.0.R : z \leq s) \wedge (\forall z : z \in s.0.L : z \geq u) \longrightarrow$
 $u.0.NB := u.0.NB / \{s\}$
grow left and *trim left* are similar

Figure 1: The bottom component of Tiara (b-Tiara).

converges to T . Since we will focus on self-stabilizing algorithms for overlay networks, and self-stabilization is only possible for overlay networks that are initially connected, we identify with **true** any state where the graph is connected.

While most of our program model is fairly conventional, we would like to draw the reader’s attention to our way of modeling overlay network link management. If one process updates its neighborhood, the change affects the neighbors of other processes. For example, if process a adds b to its neighborhood by creating a link (a, b) , this also means that a is atomically added to b ’s neighborhood. If a removes b from its neighborhood, then also a is removed from b ’s neighborhood.

3. Core Tiara Description, Correctness Proof and Complexity Estimate

In its core, Tiara contains two components: the bottom component (b-Tiara) that maintains the processes at the lowest level in sorted order and the skip-list component (s-Tiara) that constructs the higher levels of Tiara. These components are interdependent. s-Tiara relies on b-Tiara to sort the lowest level, while s-Tiara may append links to the bottom level to preserve the connectivity of the system.

We present the components and prove them correct bottom up starting with b-Tiara. However, the presentation of b-Tiara is divided into two parts: the growing and trimming. We prove the stabilization of the growing part first as the stabilization of s-Tiara depends on its correct operation. We prove the stabilization of the trimming part last as it depends on the stabilization of s-Tiara.

3.1. The Bottom Component of Tiara (b-Tiara) and Stabilization of Grow

Description. The objective of b-Tiara is to transform the system into a linear graph with the processes sorted according to their identifiers. The algorithm for b-Tiara is shown in Figure 1. The only variables that b-Tiara manipulates are the neighbor sets for each process u — $u.0.NB$. The *right neighborhood* of u , denoted $u.0.R$ is a subset of $u.0.NB$ with the identifiers greater than u . Since $u.0.R$ can be computed from $u.0.NB$ as necessary, $u.0.R$ is not an independent variable but a convenient shortcut. The *left neighborhood* $u.0.L$ is defined similarly.

Each process u has two pairs of actions: *grow* and *trim* that operate to the right and to the left of u . Action *grow right* is enabled if u discovers that its right neighbor s has a left neighbor t that is not a neighbor of u . In this case u adds t to its neighborhood. That is, u adds a link (u, t) to the graph. Even though u is the left neighbor of s , t may be either to the left or to the right of u . That is $t < u$ or $t > u$. Regardless of this relation, u connects to t . Action *grow left* operates similarly in the opposite direction.

Action *trim right* eliminates extraneous links from the graph. This action removes link (u, s) if u has a neighbor s that satisfies the following properties. The guard for *trim right* stipulates that there has to be another process t that is a neighbor of both u and s . Hence, if (u, s) is removed the connectivity of the graph is preserved. Also, all right neighbors of u must be smaller than or equal to s and all left neighbors of s are greater than or equal to u . The latter condition is necessary to break symmetry and prevent continuous growing and trimming of the same link. Action *trim left* operates similarly in the reverse direction. We show an example operation of b-Tiara in Figure 2.

Correctness proof. Denote $B(N)$ the graph that is induced by the processes of the system and the links of b-Tiara. We define the following predicate: $\mathcal{GI} \equiv (\forall a, b \in N :: \mathbf{cnsq}(a, b) \Rightarrow \exists(a, b))$. That is, \mathcal{GI} states that two consequent processes are also neighbors.

Lemma 1. *If a computation of b-Tiara starts from a state where $B(N)$ is connected, it is connected in every state of this computation.*

Proof. The actions of b-Tiara do not disconnect $B(N)$. Indeed, the actions that remove links are *trim right* and *trim left*. Consider *trim right*. It removes a link (a, b) if there exists a node c such that there are links (a, c) and (c, b) . Thus, the removal of (a, b) does not disconnect the graph. The argument for *trim left* is similar. \square

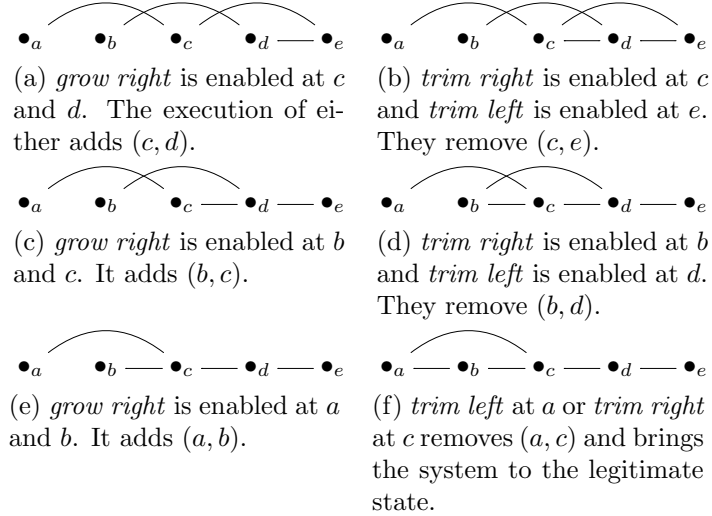


Figure 2: Example computation of b -Tiara. The processes are listed in increasing order of their identifiers.

Lemma 2. *If a computation of b -Tiara starts from a state where $B(N)$ is connected, b -Tiara stabilizes to \mathcal{GI} .*

Proof. To prove the lemma we need to show that (i) \mathcal{GI} is closed under the execution of the actions of b -Tiara and (ii) regardless of the initial state, every computation contains a state satisfying \mathcal{GI} . Let us consider closure first. The *grow* actions may not violate \mathcal{GI} as they only add links. The *trim* action may affect \mathcal{GI} by disconnecting two processes a and b . However, *trim right*, which removes link (a, b) , is only enabled at process a if there is a process c such that $a < c < b$. Therefore, if a and b are consequent, *trim right* is disabled. The reasoning is similar for *trim left*. Hence the closure.

To show convergence, let us assume that there are two consequent processes a and b that are not neighbors. That is $b \notin a.0.NB$. Since the graph itself is connected, there is a path ρ between a and b . If there are multiple paths, we shall consider the shortest one. Let the length of ρ be the sum of the lengths of its constituent links. The execution of a *trim* action does not change the length of ρ . The execution of any of the *grow* actions does not increase the length of ρ . Path ρ must contain at least one segment d, e, f such that both d and f are either smaller than e or larger than e . In this case *grow right*, or respectively, *grow left*, is enabled in both d and f . The execution of this action decreases the length of the path. Hence, throughout

the computation, the length of ρ decreases until it is zero and a and b are neighbors. The lemma follows. \square

3.2. The Skip List Component of Tiara (*s-Tiara*)

Description. The objective of *s-Tiara* is to establish a skip list on top of the linearized graph created by *b-Tiara*. The structure maintained by *s-Tiara* is a *sparse 0-1 skip list*. At each level i , node u maintains a set of neighbors $u.i.NB$. Out of this set, the rightmost and leftmost neighbors are defined as right and left skip links: $u.i.rs$ and $u.i.ls$. A node may not have a right or left skip link at some level if it is on either end of the list.

We denote right and left skip list neighbors of u at level $i - 1$ as v and x respectively. Nodes w and y are respectively right and left neighbors of v and x at the same level. We illustrate this notation in Figure 3 as we will be using it extensively throughout the correctness proof of the algorithm.

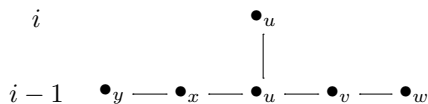


Figure 3: Aliases for neighbors of u in *s-Tiara*. $v \equiv u.(i - 1).rs$, $w \equiv v.(i - 1).rs$, $x \equiv u.(i - 1).ls$, and $y \equiv x.(i - 1).ls$, where $u.i.rs$ and $u.i.ls$ are right and left skip-list neighbors of u at level i , respectively.

If both nodes u and v exist at level i and $u.i.rs = v$ then this link is *0-skip link*. If u and w exist at level i and $u.i.rs = w$, then this link is a *1-skip link*. A process that exists at level $i - 1$ is *up* if it also exists at level i , it is *down* otherwise. If a process that 1-skip link spans is down it is a *cage*. For example u , v and w form a cage if $u.i.rs$ links to w and v is down. The middle process is *inside* the cage. Refer to Figure 4 for the illustration of the concept of a cage. The sparse 0-1 skip list has two rules of organization. First, all links are either 0 or 1 skip links. Second, if a node is on level i and it is not on the end of the list on level $i - 1$ then at least one of its links is a 1 skip link.

The the algorithm is shown in Figure 5. As before, to simplify the presentation we introduce a few shortcuts. Sets $u.i.R$ and $u.i.L$ are the subsets of $u.i.NB$ that contain the identifiers of u 's neighbors with respectively higher and lower identifiers than u . We define $u.i.rs$ to be the neighbor with the link of the smallest length among $u.i.R$. To put another way, $u.i.rs$ connects

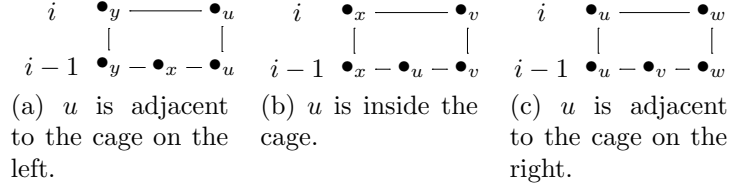


Figure 4: Possible cages with respect to node u .

to u 's right neighbor with the smallest identifier. Note that $u.i.rs$ is \perp if $u.i.R$ is empty. Shortcut $u.i.ls$ is defined similarly.

Predicate **exists**(z, i) is **true** if node z is present at all and if $z.i.NB$ is not empty. Node u may read only its immediate neighbor states. Thus, u may only invoke **exists** on its neighbors and itself. Observe that **exists** is defined to return **false** if it is invoked on a non-existent node. For example, if u is at the right end of the list at level i and u invokes **exists**($u.i.rs, i$). In this case **exists**($u.i.rs, i$) returns **false**. Predicate **valid**(u, i) captures the correct state of the system. Specifically, it states that if u exists at level i then the length of the skip links should not be more than 1 and either x or v does not exist at level i . The latter condition guarantees that at least one link of u is a 1 skip link.

The actions of s-Tiara are as follows. Action *upgrade right* establishes a link to w at level i if v is not up. That is, this link is a 1 skip link. If u is not up, *upgrade right* brings u up to level i . Action *upgrade left* operates similarly in the opposite direction. Actions *bridge right* and *left* establish 0 skip links if both nodes being connected are up. Action *prune* eliminates the links other than $u.i.rs$ and $u.i.ls$ from $u.i.NB$. In case the links are not 0 or 1 skip, action *downgrade right* completely removes the right neighborhood of u . Action *downgrade left* operates similarly. And the last action *downgrade center* eliminates three consecutive up nodes. This ensures that there could not be two consecutive 0 skip links. An example computation of s-Tiara is shown in Figure 6.

Correctness proof. Our proof proceeds as follows. We state five predicates on the level i of s-Tiara. In the sequence of lemmas we show that if the lower levels of s-Tiara have stabilized, then level i of s-Tiara stabilizes to these predicates. The conjunction of these predicates implies the stabilization of level i of s-Tiara. We then use this fact as an inductive step in the convergence

process u
parameter $i \geq 0$: **integer** — level of the skip list
variables
 $u.i.NB$ — set of neighbor processes of u at level i
shortcuts
 $v \equiv u.(i-1).rs$, $w \equiv v.(i-1).rs$, $x \equiv u.(i-1).ls$, $y \equiv x.(i-1).ls$
 $u.i.R \equiv \{z : z \in u.i.NB : z > u\}$, $u.i.L \equiv \{z : z \in u.i.NB : z < u\}$
 $u.i.rs \equiv \begin{cases} (s : s \in u.i.R : (\forall t : t \in u.i.R : t \geq s)), & \text{if } u.i.R \neq \emptyset \\ \perp, & \text{otherwise} \end{cases}$
 $u.i.ls$ is defined similarly
 $\mathbf{exists}(z, i) \equiv ((z \neq \perp) \wedge (z.i.NB \neq \emptyset))$
 $\mathbf{valid}(u, i) \equiv (((u.i.ls = y) \vee (u.i.ls = x) \vee (u.i.ls = \perp)) \wedge (u.i.rs = w)) \vee$
 $((u.i.rs = v) \vee (u.i.rs = w) \vee (u.i.rs = \perp)) \wedge (u.i.ls = y) \vee$
 $((u.i.ls = \perp) \wedge (u.i.rs = \perp)) \vee$
 $\neg(\mathbf{exists}(x, i) \wedge \mathbf{exists}(u, i) \wedge \mathbf{exists}(v, i))$
actions for $i > 0$
upgrade right: $\mathbf{valid}(u, i) \wedge \neg\mathbf{exists}(v, i) \wedge (v \neq \perp) \wedge (w \neq \perp) \wedge (u.i.rs \neq w) \longrightarrow$
 $u.i.NB := u.i.NB \cup \{w\}$
upgrade left is similar
bridge right: $\mathbf{valid}(u, i) \wedge \mathbf{exists}(u, i) \wedge \mathbf{exists}(v, i) \wedge (u.i.rs \neq v) \longrightarrow$
 $u.i.NB := u.i.NB \cup \{v\}$
bridge left is similar
prune: $\mathbf{valid}(u, i) \wedge \mathbf{exists}(u, i) \wedge (u.i.NB \neq \{u.i.rs, u.i.ls\}) \longrightarrow$
 $u.0.NB := u.0.NB \cup u.i.NB / \{u.i.rs, u.i.ls\},$
 $u.i.NB := \{u.i.rs, u.i.ls\}$
downgrade right: $\neg\mathbf{valid}(u, i) \wedge \neg((u.i.rs = v) \vee (u.i.rs = w) \vee (u.i.rs = \perp)) \longrightarrow$
 $u.0.NB := u.0.NB \cup u.i.R,$
 $u.i.R := \emptyset$
downgrade left is similar
downgrade center: $\neg\mathbf{valid}(u, i) \wedge \mathbf{exists}(x, i) \wedge \mathbf{exists}(u, i) \wedge \mathbf{exists}(v, i) \longrightarrow$
 $u.0.NB := u.0.NB \cup u.i.NB,$
 $u.i.NB := \emptyset$

Figure 5: The skip list component of Tiara (s-Tiara).

proof of stabilization of s-Tiara.

Before proceeding with the proof, we introduce notation and terminology we are going to use. Denote $S(N)$ the graph induced by the processes of the system as well as the links of b-Tiara and s-Tiara. Throughout the discussion we consider process u and its neighbors as defined in the description of s-Tiara. A node u is *middle* at level i if it has both left and right neighbors as well as at least one two hop neighbor. That is, $\mathbf{middle}(u, i) \equiv (\mathbf{exists}(v, i - 1) \wedge \mathbf{exists}(x, i - 1) \wedge (\mathbf{exists}(y, i - 1) \vee \mathbf{exists}(w, i - 1)))$.

Below are the predicates to which s-Tiara stabilizes. The **good_links.i** predicate states that process u connects to processes at most two hops away. Predicate **one_links.i** enforces the rules of 0-1 skip list. Specifically, it stipulates that u should either be inside the cage or should have adjacent cages to the left or to the right. Predicates **zero_right_links.i** and **zero_left_links.i**

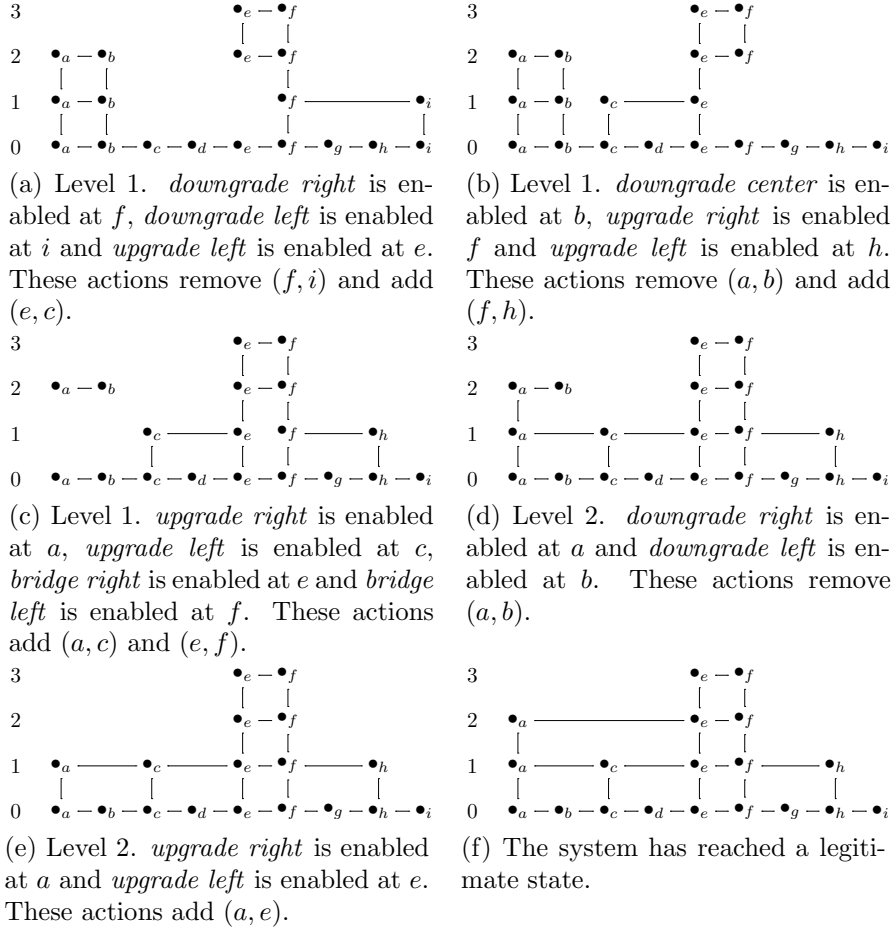


Figure 6: *s-Tiara*. We list the processes in the increasing order of their identifiers. *b-Tiara* has stabilized to \mathcal{GI} . In each state we only mention the enabled actions that are relevant to the discussion. We do not illustrate the operation of *prune*.

ensure that the 0-links are in place. That is, the processes that are consequent at level $i - 1$ and are up, are also connected at level i . Predicate **only_good_links**. i states that the neighborhood of u does not have links other than rs and ls .

$$\begin{aligned}
\mathbf{good_links}.i &\equiv (\forall u :: \neg\mathbf{exists}(u.i) \vee \\
&\quad ((u.i.rs = v) \vee (u.i.rs = w) \vee (u.i.rs = \perp)) \wedge \\
&\quad ((u.i.ls = y) \vee (u.i.rs = x) \vee (u.i.ls = \perp))) \\
\mathbf{one_links}.i &\equiv (\forall u : \mathbf{middle}(u, i) : \\
&\quad (\neg\mathbf{exists}(u, i) \wedge (x.i.rs = v) \wedge (v.i.ls = x)) \vee \\
&\quad (\neg\mathbf{exists}(v, i) \wedge (\neg\mathbf{exists}(w, i - 1) \vee (u.i.rs = w))) \vee \\
&\quad (\neg\mathbf{exists}(x, i) \wedge (\neg\mathbf{exists}(y, i - 1) \vee (u.i.ls = y)))) \\
\mathbf{zero_right_links}.i &\equiv (\forall u :: \neg\mathbf{exists}(u.i) \vee \neg\mathbf{exists}(v.i) \vee (u.i.rs = v)) \\
\mathbf{zero_left_links}.i &\equiv (\forall u :: \neg\mathbf{exists}(u.i) \vee \neg\mathbf{exists}(x.i) \vee (u.i.ls = x)) \\
\mathbf{only_good_links}.i &\equiv (\forall u :: \neg\mathbf{exists}(u.i) \vee (u.i.NB = \{u.i.rs, u.i.ls\}))
\end{aligned}$$

Lemma 3. *Assuming that neighbor relations at level $i - 1$ do not change throughout the computation, s -Tiara stabilizes to **good_links**. i*

Proof. In proving this and consequent lemmas we show a stronger property of closure and convergence of the predicate for a particular process u . This implies the stabilization of the predicate for all u at the specified level.

Let us show closure first. The topology at level $i - 1$ does not change. Hence once $u.i.rs$ points to one or two hop neighbors v or w , the neighbor's relative positions do not change. Similar argument applies to $u.i.ls$. Let us consider the actions and how they affect **good_links**. i . Let us start with the actions of u . Actions *upgrade right* and *bridge right* do not violate the predicate since they set $u.i.rs$ to respectively w and v . Similar argument applies to *upgrade left* and *bridge left*. Action *prune* does not affect the predicate since it does not modify either $u.i.rs$ or $u.i.ls$. Neither do *downgrade right* and *downgrade left* since they respectively set $u.i.rs$ and $u.i.ls$ to \perp . Action *downgrade center* removes u from level i altogether and hence cannot violate the predicate. The nodes further than two hops away never connect to u . Hence the actions of other nodes cannot violate the predicate either.

Let us now address convergence. The predicate can be violated only if u is up. It is violated if either $u.i.rs$ or $u.i.ls$ points to a node other than u 's one or two-hop neighbors. In this case either *downgrade right* or *downgrade left* are enabled that bring the links in compliance with the predicate. \square

Lemma 4. *Assuming that neighbor relations at level $i - 1$ do not change throughout the computation and **good_links.i** is satisfied, s -Tiara stabilizes to **one_links.i***

Proof. As a first step, we would like to make the following observation: once a cage is formed, it is never destroyed. For example, assume that u , v and w form a cage. The actions of u , and, similarly, w do not affect this link. Also, if v is down, the only actions it can use to come up is *upgrade right* or *upgrade left*. However, both are disabled since u and v are up. This observation guarantees the closure of **one_links.i**.

Let us discuss convergence. Assume that u is down. We consider two cases: u is initially down and u is initially up and never goes down. If u is down, the only way, u can come up is through execution of *upgrade right* or *upgrade left* at u , w or y . In all cases cages adjacent to u are formed and the predicate is satisfied. If u is down, then *upgrade right* is enabled in x and *upgrade left* in v . Thus if u does not come up, then x or v execute these *upgrade* actions. In which case a cage is formed with u inside. This satisfies the predicate as well.

Assume that u is up. If it ever goes down, the foregoing discussion applies. The only remaining case is if u stays up for the remainder of the computation. Throughout a computation of b -Tiara a node can come up only once. Indeed, a node comes up only if it forms a cage. Since a cage is never destroyed, the node never goes down. This means that a node can go down only once. Let us consider the state of the computation where u 's neighbors x and v do not change their up and down position. Both x and v cannot be simultaneously up in this state, as it enables *downgrade center* at u . The execution of this action brings u down. However, we assumed that u stays up for the remainder of the computation. Thus, either x or v are down. Assume, without loss of generality, that v is down. If w does not exist at level $i - 1$, **one_links.i** is satisfied. Assume that w exists. If link $u.i.rs = w$ is present, **one_links.i** is also satisfied. However, if it is not present, then *upgrade right* is enabled in u . Its execution establishes the link, forms a cage and satisfies the predicate. \square

Lemma 5. *Assuming that neighbor relations at level $i - 1$ do not change throughout the computation and **good_links.i** as well as **one_links.i** are satisfied, s -Tiara stabilizes to **zero_left_links.i** and **zero_right_links.i***

Proof. We prove the lemma for **zero_right_links.i** only. The proof for **zero_left_links.i** is similar. Let us argue closure. If **one_links.i** is satisfied processes do not go up or down. Thus, the only actions that can be

enabled are *bridge* and *prune*. The execution of either action maintains the validity of **zero_right_links.i**. Hence the closure.

Let us address convergence. The predicate is violated only if the neighbor processes u and v are both up and they do not have a link at level i . If **one_links.i** is satisfied, u forms a cage to its left, while v forms a cage to its right. Recall that the cages are never destroyed. In this case u has *bridge right* while v has *bridge left* enabled. When either action is executed the predicate is satisfied. \square

Lemma 6. *Assuming that neighbor relation at level $i - 1$ does not change throughout the computation and **good_links.i**, **one_links.i**, **zero_right_links.i** as well as **zero_left_links.i** are satisfied, s -Tiara stabilizes to **only_good_links.i***

Proof. (outline) The satisfaction of **good_links.i**, **one_links.i**, **zero_right_links.i** and **zero_left_links.i** leaves only one possible action enabled — *prune*. In this case there are links in $u.i.NB$ besides $u.i.rs$ and $u.i.ls$ and they are moved to $u.0.NB$. \square

Lemma 7. *If a computation of Tiara starts from a state where $S(N)$ is connected, this computation contains a state where $B(N)$ is connected.*

Proof. The non-trivial case is where $S(N)$ is connected while $B(N)$ is not. That is, the overall graph connectivity is achieved through the links at the higher levels of Tiara. Let X and Y be two graph components of $B(N)$ such that they are connected in $S(N)$. Let $i > 0$ be the lowest level where X and Y are connected. Assume, without loss of generality that there is a pair of processes $a \in X$ and $b \in Y$, such that $a.i.rs = b$. In this case *downgrade right* is enabled at a . The execution of *downgrade right* connects X and Y in $B(N)$. The lemma follows. \square

Define

$$\mathcal{SI} \equiv (\forall i : i > 0 : \mathbf{good_links.i} \wedge \mathbf{one_links.i} \wedge \mathbf{zero_right_links.i} \wedge \mathbf{zero_left_links.i} \wedge \mathbf{only_good_links.i})$$

Lemma 8. *Tiara stabilizes to \mathcal{SI} .*

Proof. According to Lemma 7, every computation contains a state where $B(N)$ is connected. Due to Lemma 2, if $B(N)$ is connected, b -Tiara stabilizes to \mathcal{GI} . The remainder of the proof is by induction on the levels of

s-Tiara. If $B(N)$ is connected and \mathcal{GI} is satisfied the topology of the level 0 does not change. Hence, the requisite five predicates are vacuously satisfied. Assume that these predicates are satisfied for all levels $i - 1$. Once the predicates are satisfied, none of the actions for processes at level $i - 1$ are enabled. This means that the topology at this level does not change. Applying Lemmas 3, 4 5 and 6 in sequence we establish that the five predicates are satisfied at level i . Hence the lemma. \square

3.3. Stabilization of Trim in b-Tiara

Link (a, b) is *independent* if there exists no link (c, d) different from (a, b) such that $c \leq a$ and $b \leq d$. Consider an arrangement where the nodes are positioned in the increasing order of their identifiers.

Lemma 9. *If a computation of b-Tiara that starts in a state where the graph is connected and contains an independent link of non-zero length, this computation also contains a suffix of states without this link.*

Proof. Let (a, b) be an independent link of non-zero length. None of the *grow* actions create independent links. The only action that makes a link independent is a *trim* of another independent link. Thus, if an independent link is deleted, it is never added. Thus, to prove the lemma it is sufficient to show that (a, b) is eventually deleted.

Link (a, b) is non-zero length. This means that the node c consequent to a is not the same as b . In other words $a < c < b$. b-Tiara stabilizes to \mathcal{GI} which ensures that a and c are connected. If c and b are not connected, both of them have a *grow* action enabled that connects them. Observe that (a, b) is independent. This means that all the right neighbors of a are to the left of b and all the left neighbors of b are to the right of a . Moreover, we just showed that there exists a node c such that $a < c < b$ and there are links $c \in a.R$ and $c \in b.L$. This means that *trim right* is enabled at a and *trim left* is enabled at b . The execution of either action deletes (a, b) . \square

We define the following predicate:

$$\mathcal{TI} \equiv (\forall a, b \in N :: \exists(a, b) \Rightarrow \mathbf{cnsq}(a, b))$$

Lemma 10. *If Tiara starts in a state where it satisfies \mathcal{GI} and \mathcal{SI} , then it stabilizes to \mathcal{TI}*

Proof. (outline) The conjunct of \mathcal{GI} and \mathcal{TI} is closed under the execution of b-Tiara. Note also that if \mathcal{GI} and \mathcal{SI} are satisfied, then the actions s-Tiara are disabled. Hence the closure of \mathcal{TI} .

Let us consider convergence. Since the actions of s-Tiara are disabled, they do not add links to $B(N)$. If \mathcal{TI} does not hold, then there is at least one independent link of non-zero length. If the graph is connected the *grow* actions never create an independent link. Consider a computation of b-Tiara that starts in an illegitimate state. Let l be the length of the longest independent link. Since the state is not legitimate, $l > 0$. According to previous discussion, new links of length l do not appear. Let (a, b) be the independent link of length l . According to Lemma 9, (a, b) is eventually removed. Thus, all links of length l are eventually removed. The lemma can be easily proven by induction on l . \square

The discussion in this section culminates in the following theorem.

Theorem 1. *Tiara stabilizes to the conjunction of \mathcal{GI} , \mathcal{SI} and \mathcal{TI} .*

4. Searches and Updates in Tiara

Searches. Tiara maintains a skip list [10, 11] which is equivalent to a distributed balanced search tree. Hence the searches in Tiara proceed similar to searches in such trees. Let b be a right neighbor of a at some level i of Tiara. The *right interval* of a , denoted $[a, b)$, is the range of identifiers between a and b . *Left interval* is defined similarly. If a does not have a right neighbor, its interval is not finite. That is, a 's interval contains all process identifiers greater than a . Similarly, if a lacks left neighbor its interval is infinite on the left. Thus in any level, the collection of intervals contains the complete range of identifiers.

Suppose a , c and b are consequent at level $i - 1$ of Tiara and a and b are consequent at level i . That is c is in the cage. Since the identifiers are sorted, c belongs to the interval $[a, b)$. If a node is down, then one of its neighbors is up. Thus a client process that has a pointer to a node in Tiara and wishing to advance up the skip list only needs to examine the node's neighbors.

Assuming that a client process connects to an arbitrary node in Tiara, the search proceeds first upward then downward in the skip list. In the upward phase, the client is moving up the list looking for the node whose interval contains the identity. Since every level contains the complete id-range, this phase terminates. Once the range is found, the client advances downward

evaluating the cages it encounters to narrow the search range. This procedure continues until the desired node x is located or it is established that x belongs to the interval of the consequent nodes at the bottom level. The latter case means that x is not present in the system. There are $O(\log|N|)$ levels in Tiara. Thus, the upward and the downward phases take $O(\log|N|)$ number of steps.

Joins and leaves. We assume that each process has two read-only Boolean variables maintained by the environment: *join* and *leave*. Since the variables are read-only, stabilization of their operation is the responsibility of the environment. Let us consider join operation first. The joining node x connects to an arbitrary node of the network. The variable *join* is set to **true**. We assume that the environment may only set *join* to **false** after the node successfully inserts itself at the bottom level of Tiara. The joining node executes a search to find the bottom level interval $[a, b)$ to which it belongs. Then, x makes a and b its right and left neighbors respectively. After a and b discover the presence of a node whose *join* is set to **true**, they remove link (a, b) . Then, the upper levels of Tiara adjust. The insertion of the node at the bottom level entails at most a constant number of steps at each level of Tiara. Since the search takes at most $O(\log|N|)$ steps, the total number of steps required for node join is also in $O(\log|N|)$.

Let us discuss the leave operation. The environment sets *leave* to **true** to indicate that the node x requests disconnect. We assume that *leave* cannot be set when *join* is set and it cannot be set back to **false** until the node disconnects. When the right and left neighbors of x notice that the *leave* of x is set to **true**, the neighbors add a link bypassing x at the bottom level. Node x can then disconnect. The higher levels of Tiara execute the regular Tiara actions to accommodate the missing node. At most a constant number of adjustment steps is required at each level. Hence the total number of steps required for the node to leave Tiara is in $O(\log|N|)$.

5. Extension to a Ring (r-Tiara)

Outline. Tiara can be extended to a ring structure similar to Chord [9]. We call this extension r-Tiara. The idea is as follows. For b-Tiara, as well as for each level of s-Tiara, the lowest id-process needs to add a special wraparound link to the highest-id process. This wraparound link maintenance is carried out by the process without left neighbors. After b-Tiara and s-Tiara stabilize,

the lowest-id process at each level is the only such process. The highest-id process at each level is the only process without right neighbors.

Once the process determines that it has no left neighbors it starts positioning the wraparound link. Essentially, the process continues to move the link to a right neighbor of the destination of the link. This movement stops once the wraparound link reaches the highest-id process at that level. If the maintainer of the wraparound link determines that it has left neighbors, it destroys its wraparound link.

```

process  $u$ 
variables
   $u.0.NB$ , set of neighbor processes of  $u$ .
   $u.0.WA$ , wraparound link set
shortcuts
   $u.0.L \equiv \{s : s \in u.0.NB : s < u\}$ ,  $u.0.R \equiv \{s : s \in u.0.NB : s > u\}$ 
actions
 $wrap$ :  $(u.0.L = \emptyset) \wedge (u.0.WA = \emptyset) \wedge (u.0.R \neq \emptyset) \longrightarrow$ 
         $u.0.WA := \{s | s \in u.0.R : (\forall z : z \in u.0.R : z \leq s)\}$ 
 $extend$ :  $(u.0.L = \emptyset) \wedge (t \in u.0.WA \cup u.0.R) \wedge$ 
         $(s, t : (s \in t.0.R \cup u.0.R) : (\forall z : z \in u.0.WA : s > z)) \longrightarrow$ 
         $u.0.WA := u.0.WA \cup \{s\}$ 
 $purge$ :  $(u.0.L = \emptyset) \wedge (s : s \in u.0.WA : (\exists z : z \in u.0.WA : u < s < z)) \longrightarrow$ 
         $u.0.NB := u.0.NB \cup \{s\},$ 
         $u.0.WA := u.0.WA / \{s\}$ 
 $expunge$ :  $(u.0.L \neq \emptyset) \wedge (s : s \in u.0.WA : s > u) \longrightarrow$ 
         $u.0.NB := u.0.NB \cup \{s\},$ 
         $u.0.WA := u.0.WA / \{s\}$ 

```

Figure 7: The ring component of Tiara (r-Tiara)

Description. We describe the operation of r-Tiara at the bottom level, although it would execute on all levels. The objective of r-Tiara is to construct a ring on the level. This ring is established by connecting the lowest-id process on the level to the highest-id process on the level, which is respectively $\min(N)$ and $\max(N)$ on the bottom level. The code for r-Tiara is shown in Figure 7. In addition to the neighbor set, r-Tiara maintains a wraparound set ($u.0.WA$). Each process u has four actions: *wrap*, *extend*, *purge* and *expunge*. Action *wrap* is enabled at the lowest-id process. Specifically, *wrap* is enabled if u does not have left neighbors, has a right neighbor, and currently is not participating in a wraparound link. In this case, u adds the rightmost process it has in its neighborhood to a wraparound variable. Action *extend* grows the wraparound link towards $\max(N)$ on the bottom level. This action adds to the wraparound set a neighbor of u or a neighbor of a neighbor of u that

has an identifier greater than any process id already in the wraparound set. When superfluous links exist in the wraparound variable, they are removed by the *purge* action. In the case where u gains a left neighbor and is currently maintaining the wraparound variable, action *expunge* removes all links to the right of u from the wraparound variable. Both *purge* and *expunge* move links to the neighbor variable to prevent the possibility of partitioning the graph. The operation of r-Tiara on the bottom level is illustrated in Figure 8.

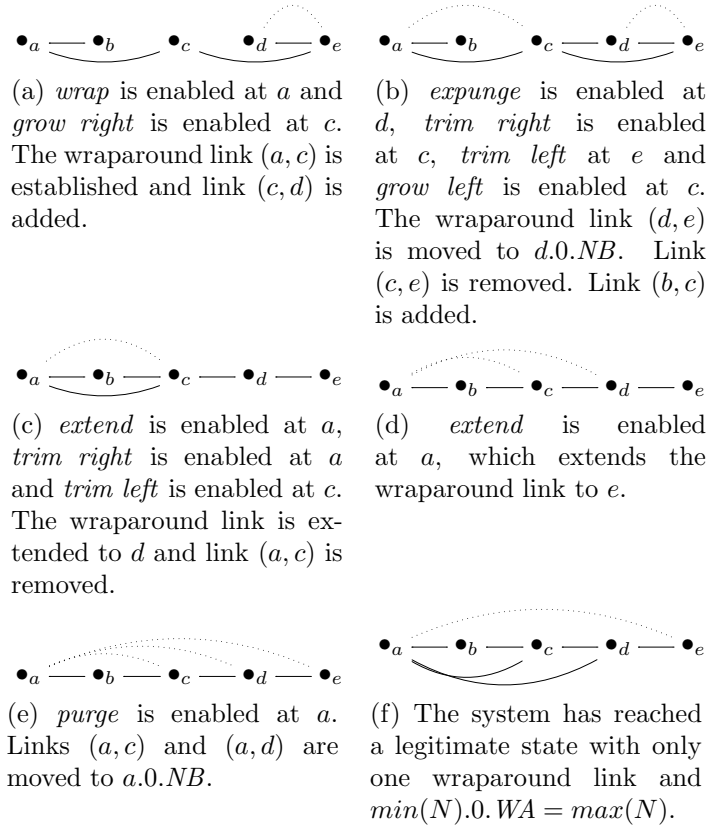


Figure 8: Example computation of *r-Tiara*. The processes are listed in increasing order of their identifiers. Not all enabled actions are listed.

Correctness proof. We denote $R(N)$ the graph induced by the processes, the b-Tiara links (the level 0 s-Tiara links) and the wraparound links. We

define the following predicate

$$\begin{aligned} \mathcal{RI} \equiv & (\min(N).0.WA = \max(N)) \wedge \\ & (\forall a \in N : (a \neq \min(N)) \wedge (a \neq \max(N))) : a.0.WA = \emptyset \end{aligned}$$

That is, \mathcal{RI} states that the only wraparound link that the bottom level has connects the nodes with the largest and smallest identifiers.

Lemma 11. *If a computation of b-Tiara, s-Tiara and r-Tiara starts from a state where $B(N)$ is connected, it stabilizes to $\mathcal{GI} \wedge \mathcal{SI} \wedge \mathcal{RI}$.*

Proof. Let us address the closure first. We have to consider the influence of the actions of r-Tiara on \mathcal{GI} since some of them modify the variables of b-Tiara. Fortunately, *purge* and *expunge* are the only such actions. Moreover, they cannot invalidate \mathcal{GI} since they only add links to the neighborhood of a node and do not remove them. The actions of r-Tiara do not influence \mathcal{SI} . Let us attend to the closure of \mathcal{RI} . If $\mathcal{GI} \wedge \mathcal{SI} \wedge \mathcal{RI}$ is satisfied, the only node on the bottom level that has an empty left neighborhood is $a = \min(N)$. Thus, only a may have *wrap* and *extend* enabled. Yet, \mathcal{RI} stipulates that the wraparound variable of a is not empty. Thus, *wrap* is disabled when \mathcal{RI} holds. Also if \mathcal{RI} holds, the wraparound link connects a and $b = \max(N)$. Node b has an empty right neighborhood ($b.0.R$). Thus, *extend* is also disabled at a . Let us discuss *expunge*. If \mathcal{RI} holds, b is the only node whose left neighborhood is not empty yet $b.0.WA \neq \emptyset$. However, for b , $b.0.WA = a$ which is less than b . Hence, *expunge* is disabled as well. Action *purge* is disabled by definition if \mathcal{RI} holds.

Let us now address the convergence of the predicates. r-Tiara does not affect the convergence of \mathcal{GI} or \mathcal{SI} as it can only add links to a node's neighborhood but not remove them. Thus, we can prove the convergence of \mathcal{RI} while $\mathcal{GI} \wedge \mathcal{SI}$ is satisfied. If $a = \min(N)$ on the bottom level does not have a wraparound link, *wrap* is enabled. The execution of *wrap* creates this link. No other wraparound links can be created. That is, if a computation starts from a state where a has a wraparound link, the number of wraparound links in the states of this computation can only decrease. Consider a wraparound link whose left (smaller) incident link is not a . *expunge* is enabled at this node. The execution of *expunge* removes such link. That is, each computation contains a suffix where a is incident to the only existing wraparound link. If $b = \max(N)$ on the bottom level is not incident to this link, *extend* is enabled at a . If *extend* is executed the identifier of the right incident node

increases. Thus, the computation contains a state where b is incident to this link. If $b \in a.0.WA$ and $a.0.WA \neq b$, then *purge* is enabled at a . When executed only b will remain in $a.0.WA$.

That is, if $\mathcal{GI} \wedge \mathcal{SI}$ is satisfied, a computation of r-Tiara contains a suffix where in each state there is a single wraparound link that connects processes with the minimum and maximum ids on the level. That is r-Tiara converges to \mathcal{RI} . \square

Lemma 12. *If a computation of b-Tiara, s-Tiara and r-Tiara starts from a state where $R(N)$ is connected, this computation contains a state where $B(N)$ and $S(N)$ are connected.*

Proof. (outline) The only non-trivial case is where the computation of b-Tiara, s-Tiara and r-Tiara start from a state in which $R(N)$ is connected while $B(N)$ and $S(N)$ are not. That is, the overall graph connectivity is achieved via wraparound links.

Let X and Y be two disconnected graph components of $B(N)$ and $S(N)$ such that there is a pair of processes $a \in X$ and $b \in Y$ and $a.0.WA = b$. Let us consider each component as a separate system. According to Lemma 11, r-Tiara arrives at a state where the only existing wraparound link connects the processes with the smallest and largest identifiers in each component. That is link (a, b) is no longer a wraparound link. r-Tiara does not delete the wraparound links. Instead, it moves them to $B(N)$. Thus, the computation contains a state where the (a, b) link belongs to $B(N)$. That is, the graph components X and Y are connected. The lemma follows. \square

Lemmas 11 and 12 combined yield the following lemma

Lemma 13. *If a computation of b-Tiara and s-Tiara with r-Tiara starts from a state where $R(N)$ is connected, it stabilizes to $\mathcal{GI} \wedge \mathcal{SI} \wedge \mathcal{RI}$.*

6. Extension to Skip Graph (g-Tiara)

Outline. The disadvantage of using skip lists for peer-to-peer systems is their high congestion. Specifically, a large number of requests have to go through the nodes at the highest level of the skip-list. These nodes become a bottleneck. Moreover, the failure of a single node disconnects a skip list without the ring extension. To mitigate these problems we propose to extend Tiara to concurrently construct multiple skip lists forming a skip graph. We call this component g-Tiara.

The idea is as follows. Recall that at each level $i - 1$, core Tiara does not upgrade at least one-third of the nodes. The upgraded nodes form a list at level i while the remainder do not form any links in higher levels. In g-Tiara, the remaining nodes form their own list at level i' . This list is used to concurrently run the next level of Tiara. In our description we ignore the special cases that arise at the edges of the skip graph because they turn out to be inconsequential. We also ignore the operation of r-Tiara at each level.

Description. The code of g-Tiara is shown in Figure 9. We assume that b-Tiara is in operation. That is, b-Tiara links the processes with consecutive identifiers at level 0. At each level $i > 0$, s-Tiara forms a list of nodes upgraded from level $i - 1$. g-Tiara does not interfere with the operation of s-Tiara but uses this list to construct the alternative list of nodes that are not upgraded by s-Tiara.

Recall that a process is down at a level i if it is not upgraded by s-Tiara. A down process can be at most two hops away from another down process at level $i - 1$. To locate the nearest down neighbor, in g-Tiara each process u at level i , maintains a set $u.i.NB2$ of two hop neighbors. The set of alternative neighbors is collected in $u.i.NB'$. This set contains the down processes that are at most three hops away in level $i - 1$.

g-Tiara has two actions. Action *gather* maintains $u.i.NB2$ on the basis of the information stored in single-hop neighbors. Action *connect* updates $u.i.NB'$. Note that even though *gather* shows the construction of the neighborhood at level i , *connect* uses two neighborhood information at level $i - 1$.

To avoid partitioning, the incorrect links are moved to $u.0.NB$. According to the execution semantics, once a process z is added to one of the sets maintained by u , for example $u.i.NB'$. Process u is also added to the equivalent set in z , that is $z.i.NB'$.

Let us examine the neighborhood $u.i.NB$ that is formed by each node u upgraded by s-Tiara. The links to the closest left and right neighbors of u form a list of nodes at level i . This list is used to construct $u.(i + 1).NB$ by s-Tiara and $u.(i + 1).NB'$ by g-Tiara. Similarly, on the basis of $u.i.NB'$, an alternative list of nodes not upgraded by s-Tiara is formed. Separate instances of s-Tiara and g-Tiara run on the alternative list and construct another pair of lists on level $i + 1$ as well. Refer to Figure 10 for an illustration of a complete skip graph built by this extension to core Tiara.

Correctness proof. The operation of the algorithm is rather straightforward.

```

process  $u$ 
parameter  $i \geq 0$ : integer — level of the skip graph
constant  $u.i.NB$  — set of neighbor processes of  $u$  at level  $i$ 
variables
   $u.i.NB2$  — 2-hop neighborhood set variable of process  $u$  at level  $i$ 
   $u.i.NB'$  — alternative neighbors of  $u$ 
shortcuts
   $\text{exists}(z, i) \equiv ((z \neq \perp) \wedge (z.i.NB \neq \emptyset))$ 
   $u.i.NB2DST \equiv \{s : (\forall t : t \in u.i.NB : s \in (u.i.NB \cup t.i.NB))\}$ 
   $u.i.DOWN \equiv \{z : z \in \{s : (\forall t : t \in u.(i-1).NB2 : s \in (u.(i-1).NB2 \cup t.(i-1).NB))\} : \neg \text{exists}(z, i)\}$ 
actions
for  $i \geq 0$ 
  gather:  $u.i.NB2 \neq u.i.NB2DST \longrightarrow$ 
     $u.0.NB := u.0.NB \cup u.i.NB2 / u.i.NB2DST,$ 
     $u.i.NB2 := u.i.NB2DST$ 
for  $i > 0$ 
  connect:  $u.i.NB' \neq u.i.DOWN \longrightarrow$ 
     $u.0.NB := u.0.NB \cup u.i.NB' / u.i.DOWN,$ 
     $u.i.NB' := u.i.DOWN$ 

```

Figure 9: The skip graph component of Tiara (g-Tiara).

ward. We, therefore, present the informal correctness statement in the below theorem and show the proof outline.

Theorem 2. *At each level i , for each down process u , the nearest left and right down neighbors of u are in the alternative neighborhood set $u.i.NB'$.*

Proof. (outline) The proof is by induction. The bottom level is maintained by b-Tiara. This level stabilizes to \mathcal{GI} regardless of actions of g-Tiara. Thus, the list of the processes at the bottom level eventually remains unchanged. This allows r-Tiara to stabilize to \mathcal{RI} at level 0. Assume that the list of processes and the corresponding wraparound link does not change at level $i - 1$. The construction of s-Tiara list at level i proceeds independently of g-Tiara. After s-Tiara stabilizes to \mathcal{SI} , the list at level i is a sparse 0 – 1 skip list. This means that for each down node the nearest neighbor is at most three hops away. Meanwhile, due to action *gather* of g-Tiara, for each process u , $u.(i - 1).NB2$ contains correct two hop neighborhood information. After this information is stable, *connect* adds down processes up to three hops away from u to $u.i.NB'$. The theorem follows. \square

Complexity Estimate. The skip graph can be viewed as a collection of skip-lists rooted in every node. Thus, unlike a skip list, the search in a skip

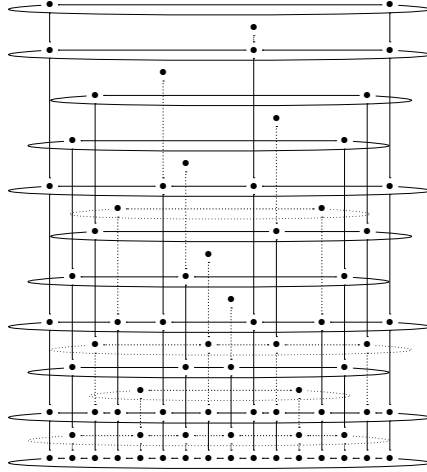


Figure 10: Example skip graph produced by g-Tiara.

graph has downward phase only. Therefore, the number of steps required to perform a search in a skip graph is in $O(\log|N|)$.

The complexity estimate for join or leave of a node is more involved.

Lemma 14. *A join of a node at level $i - 1$ requires a constant number of steps, a join of a node in one of the resultant lists at level i and possibly a change of a single node in the other list. Similarly, a leave of a node at level $i - 1$ requires a constant number of steps, a leave of a node in one of the resultant lists at level i and a change of a single node in the other list.*

Proof. We discuss node join. The argument for node leave is similar. Let node z join the neighborhood of node u at level $i - 1$. Let the neighbor aliases be as shown in Figure 3.

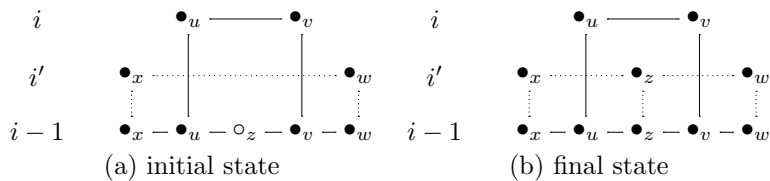


Figure 11: Node z joins the skip graph at level $i - 1$ under the bridge of the s-Tiara list.

A node may join under the bridge of s-Tiara or inside a cage. Let us consider the bridge first. See Figure 11 for illustration. In this case, the

actions of s-Tiara are disabled and the only outcome is for z to be added to the list maintained by g-Tiara at level i' . This requires two actions of g-Tiara. That is, one of the sublists does not change while a node joins the other list.

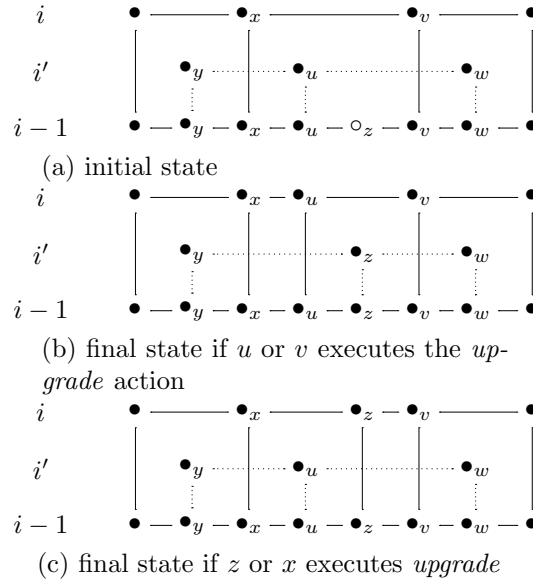


Figure 12: Node z joins the skip graph inside a cage with no adjacent bridges.

Let us now consider the case of z joining inside a cage of s-Tiara. The outcome differs depending on whether there are bridges adjacent to this cage. Let us discuss the subcase of no adjacent bridges. Refer to Figure 12. The appearance of z enables actions of s-Tiara at x , u , z and v . The execution of *downgrade* actions at x or v does not affect the final state. The final state depends on which process executes an *upgrade* action. If process u or v executes *upgrade*, u joins the s-Tiara list at level i while z replaces u in g-Tiara list. If process x or z executes *upgrade*, z joins the s-Tiara list while, g-Tiara list is not affected. In either case the conditions of the lemma are satisfied.

The last case to consider is where z joins a cage and there is an adjacent bridge. We only discuss the case where there is a single adjacent bridge to the right of the cage. The other cases are similar. Refer to Figure 13. Again, the final state depends on which process executes *upgrade*. If u or v execute this action, it results in the formation of the second bridge, u joining the

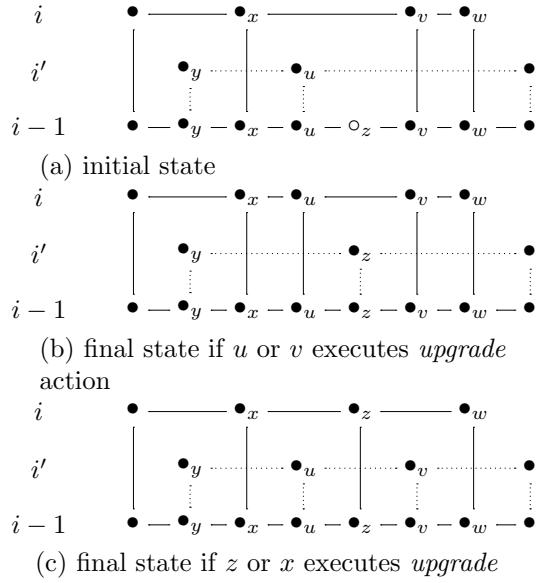


Figure 13: Node z joins the skip graph inside a cage with adjacent bridge to the right.

s-Tiara list and z replacing u in the g-Tiara list. However, if z or x execute *upgrade*, three consequent nodes z , v and w are upgraded to level i from level $i - 1$ in s-Tiara. This forces v to execute *downgrade center*. The final state results in v joining the g-Tiara list while z replacing v in s-Tiara. This satisfies the conditions of the lemma. \square

We calculate the complexity of a topology update based on Lemma 14. According to the lemma, the topology update requires a constant number of steps and results in a similar topology update in one of the sublists and a node replacement in the other. Let c be the constant upper bound on the number of steps required for this propagation to occur.

Single member replacement then requires c number of steps at each level in one of the sublists. In other words, the propagation of the update to a single level i results in the addition of $i \cdot c$ steps. There are at most $\log|N|$ levels in the skip graph. Thus, the total number of steps required to add or remove a node to g-Tiara skip list is:

$$\sum_{i=0}^{\log|N|} i \cdot c = c \frac{\log|N| \cdot (\log|N| + 1)}{2} = O(\log^2|N|)$$

Congestion and expansion bounds. In a *random routing problem*, every node in the network has exactly one message for a node chosen uniformly at random [3, 22]. Given a routing strategy R in some graph G , the *congestion* of a fixed routing problem is the maximum number of messages traversing a node when using R for that problem. When using a random routing problem, we are interested in the expected congestion. In a complete binary tree, for example, the expected congestion is $\Theta(|N|)$ as, on expectation, half of the nodes below the left son of the root want to send their message to one of the nodes below the right son of the root, and vice versa. Hence, $\Theta(|N|)$ messages have to cross the root on expectation, no matter which routing strategy is used for them. The same bound holds for the skip list since the removal of the root node in the skip list cuts it into two connected components of approximately the same size. Thus, any message from one of these components to the other has to cross the root node. A much better congestion can be achieved for the skip graph.

In our g-Tiara skip graph, any edge in a level i -ring connects nodes of distance at most 3 in the $(i - 1)$ -ring and no three consecutive nodes belong to the same i -ring. Hence, any edge of an i -ring can skip at most 3^i nodes on the base ring, and any i -ring contains at most $(2/3)^i \cdot |N|$ nodes. Consider the routing strategy of routing every request to its destination in a top-down fashion: in each hop, the highest level edge towards the destination is used without getting beyond the destination. Then every request traversing an i -level edge must originate from a node in that i -ring. Therefore, the expected number of requests passing a node at level i is at most

$$\left(\frac{2}{3}\right)^i \cdot |N| \cdot \frac{3^i}{|N|} = 2^i$$

We can only have that many requests if $2^i \leq (2/3)^i \cdot |N|$ (the number of nodes in level i), which is true so long as $i \leq \log_3 \cdot |N|$. Therefore, the expected congestion is at most $2^{\log_3 \cdot |N|} = |N|^{1/\log 3}$.

With the help of the expected congestion, we can find a lower bound on the expansion of a graph. Let U be an arbitrary subset of nodes in the given graph. A neighbor set $NB(U)$ is defined as $\{w \in V \setminus U \mid \exists v \in U : (v, w) \in E\}$. That is, a node belongs to $NB(U)$ if it does not belong to U but is a neighbor of at least one node of U . The *(node) expansion* of a graph is defined as $\min_{U \subset N, |U| \leq |N|/2} |NB(U)|/|U|$. The expansion captures the degree

of connectivity of a graph, and the best expansion of a constant degree graph is constant.

Given a congestion of α , it is possible to design a permutation routing problem (every node is the source and destination of exactly one message) so that there is a node that is passed by at least $1/\alpha$ many messages, no matter which routing strategy is used (all messages from the set U with $|NB(U)| = \alpha|U|$ are requested to leave U). On the other hand, every permutation routing problem can be solved via two random routing problems: first, route each message to a random intermediate destination and then from there to its final destination. In fact, when using this strategy, the expected congestion for routing any permutation routing problem is twice the expected congestion of a random routing problem. Thus, if a random routing problem can be routed with expected congestion C , any permutation routing problem can be routed with expected congestion $2C$, which implies that for any permutation routing problem there is a routing strategy with congestion at most $3C$. This, however, is only possible if the expansion of the given graph is at least $1/3C$. Thus, the g-Tiara has an expansion of $\Omega(|N|^{-1/\log 3})$, which is much better than the expansion $\Theta(1/|N|)$ of the tree and the skip list.

7. Implementation and Extensions

Crash resistance. Tiara can be separated into disconnected components by the crash of even a single process. Tiara can be fortified against separation due to crashes in the following manner. At the bottom, each process maintains a crash-redundancy link to its right neighbor's neighbor. That is, the bottom level list becomes doubly connected. Thus, it can tolerate a single crash. The crash tolerance can be further improved by adding similar links to more distant processes. In an asynchronous model there is no reliable way to distinguish a crashed process from a slow one [23]. Thus, to accomplish this, the processes need to be equipped with failure detectors [24, 25]. A failure detector alerts the process if its neighbor crashes. Then, Tiara stabilizes to a legitimate state corresponding to the system without the crashed process.

Other improvements. There is a number of modifications to Tiara that make it more efficient and applicable. At each level of Tiara, up to two out of three nodes may be promoted to the next level. Although the number of levels is logarithmic with respect to the system size, it may still be relatively large. The number of levels may be decreased by modifying Tiara to promote

fewer nodes. For example, we can allow the nodes at level i to skip up to two or three neighbors at level $i - 1$. This would require for each node to maintain data about its extended neighborhood.

The *grow* operation of b-Tiara may force a process to acquire up to $O(|N|)$ neighbors during stabilization. This may require devoting extensive memory resources of each node to neighborhood maintenance. A simple way to mitigate it is to execute *trim* operations before *grow*. That is, if a process finds that it has both *trim* and *grow* actions enabled. It executes *trim*. Care must be taken to ensure that action execution is still weakly fair.

8. Future Work

We presented Tiara — a first deterministic self-stabilizing peer-to-peer system with a logarithmic diameter. It provides a blueprint for a realistic system. An important further task is to study the implementation of Tiara in more realistic low-atomicity models such as message passing. We envision several other directions of extending this work: further efficiency improvements, such as keeping the runtime and the degree of the self-stabilization process low, and adding features required by practical systems. An important property is resistance to *churn* — continuous leaving and joining of nodes. Improving Tiara’s resistance to churn is a significant avenue of future research.

References

- [1] D. Andersen, H. Balakrishnan, F. Kaashoek, R. Morris, Resilient overlay networks, in: SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles, ACM, New York, NY, USA, 2001, pp. 131–145. doi:<http://doi.acm.org/10.1145/502034.502048>.
- [2] J. Aspnes, G. Shah, Skip graphs, in: SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003, pp. 384–393.
- [3] B. Awerbuch, C. Scheideler, The hyperring: a low-congestion deterministic data structure for distributed environments, in: SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete

algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004, pp. 318–327.

- [4] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, M. Thober, Pagoda: a dynamic overlay network for routing, data management, and multicasting, in: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, ACM, New York, NY, USA, 2004, pp. 170–179. doi:<http://doi.acm.org/10.1145/1007912.1007938>.
- [5] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, A. Wolman, Skipnet: a scalable overlay network with practical locality properties, in: USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, USENIX Association, Berkeley, CA, USA, 2003, pp. 9–9.
- [6] D. Malkhi, M. Naor, D. Ratajczak, Viceroy: a scalable and dynamic emulation of the butterfly, in: PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM, New York, NY, USA, 2002, pp. 183–192. doi:<http://doi.acm.org/10.1145/571825.571857>.
- [7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, A scalable content-addressable network, in: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, New York, NY, USA, 2001, pp. 161–172. doi:<http://doi.acm.org/10.1145/383059.383072>.
- [8] A. I. T. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Springer-Verlag, London, UK, 2001, pp. 329–350.
- [9] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup protocol for internet applications, *IEEE/ACM Trans. Netw.* 11 (1) (2003) 17–32.

- [10] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, *Commun. ACM* 33 (6) (1990) 668–676. doi:<http://doi.acm.org/10.1145/78973.78977>.
- [11] J. I. Munro, T. Papadakis, R. Sedgewick, Deterministic skip lists, in: *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992, pp. 367–375.
- [12] B. Awerbuch, C. Scheideler, Group spreading: A protocol for provably secure distributed name service, in: J. Díaz, J. Karhumäki, A. Lepistö, D. Sannella (Eds.), *ICALP*, Vol. 3142 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 183–195.
- [13] L. O. Alima, S. Haridi, A. Ghodsi, S. El-Ansary, P. Brand, Position paper: Self-properties in distributed k-ary structured overlay networks, in: *Proceedings of SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, Vol. 3460 of *Lecture Notes in Computer Science*, Springer, 2004.
- [14] M. Onus, A. W. Richa, C. Scheideler, Linearization: Locally self-stabilizing sorting in graphs, in: *ALLENEX 2007: Proceedings of the Workshop on Algorithm Engineering and Experiments*, SIAM, 2007.
- [15] A. Shaker, D. S. Reeves, Self-stabilizing structured ring topology p2p systems, in: *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, IEEE Computer Society, Washington, DC, USA, 2005, pp. 39–46. doi:<http://dx.doi.org/10.1109/P2P.2005.34>.
- [16] T. Héroult, P. Lemarinier, O. Peres, L. Pilard, J. Beauquier, Brief announcement: Self-stabilizing spanning tree algorithm for large scale systems, in: A. K. Datta, M. Gradinariu (Eds.), *SSS*, Vol. 4280 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 574–575.
- [17] C. Cramer, T. Fuhrmann, Isprp: a message-efficient protocol for initializing structured p2p networks, in: *IPCCC 2005: Proceedings of the 24th IEEE International Performance Computing and Communications Conference*, IEEE, 2005, pp. 365–370.

- [18] E. Caron, F. Desprez, F. Petit, C. Tedeschi, Snap-stabilizing prefix tree for peer-to-peer systems, in: T. Masuzawa, S. Tixeuil (Eds.), SSS, Vol. 4838 of Lecture Notes in Computer Science, Springer, 2007, pp. 82–96.
- [19] S. Bianchi, A. Datta, P. Felber, M. Gradinariu, Stabilizing peer-to-peer spatial filters, in: ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2007, p. 27. doi:<http://dx.doi.org/10.1109/ICDCS.2007.139>.
- [20] S. Dolev, R. I. Kat, Hypertree for self-stabilizing peer-to-peer systems, *Distributed Computing* 20 (5) (2008) 375–388.
- [21] D. Dolev, E. Hoch, R. van Renesse, Self-stabilizing and byzantine-tolerant overlay network, in: OPODIS 2007: Proceedings of the 11th International Conference on the Principles of Distributed Systems, Vol. 4878 of Lecture Notes in Computer Science, Springer, 2007, pp. 343–357.
- [22] C. Scheideler, *Universal Routing Strategies for Interconnection Networks*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
- [23] M. J. Fischer, N. A. Lynch, M. S. Paterson, Impossibility of distributed consensus with one faulty process, *J. ACM* 32 (2) (1985) 374–382. doi:<http://doi.acm.org/10.1145/3149.214121>.
- [24] T. D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, *J. ACM* 43 (4) (1996) 685–722. doi:<http://doi.acm.org/10.1145/234533.234549>.
- [25] T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, *J. ACM* 43 (2) (1996) 225–267. doi:<http://doi.acm.org/10.1145/226643.226647>.