A Model for Large Scale Self-Stabilization [*][†]

Thomas Herault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, Joffroy Beauquier
Univ Paris Sud; LRI UMR8623; INRIA; Orsay F-91405
{herault,lemarini,peres,pilard,jb}@lri.fr

Abstract

We introduce a new model for distributed algorithms designed for large scale systems that need a low-overhead solution to allow the processes to communicate with each other. We assume that every process can communicate with any other process provided it knows its identifier, which is usually the case in e.g. a peer to peer system, and that nodes may arrive or leave at any time. To cope with the large number of processes, we limit the memory usage of each process to a small constant number of variables, combining this with previous results concerning failure detectors and resource discovery. We illustrate the model with a self-stabilizing algorithm that builds and maintains a spanning tree topology. We provide a formal proof of the algorithm and the results of experiments on a cluster.

Keywords: Distributed Algorithm, Self-Stabilization, Large Scale Systems, Spanning Tree Construction, Failure Detectors.

1 Introduction

Peer to peer networks and grids are emerging large scale systems that gather thousands of nodes. These networks usually rely on IP to communicate: each node has a unique address used by other nodes to communicate with it.

Usually, self-stabilizing algorithms are designed for distributed systems defined by their topology. Each process has a finite set of communication links to exchange messages with its neighbors. In our model, we replace the existence of a complete topology with the notion of neighborhood, based on resource discovery. No process knows the set of its links and, since this set is very large, no process attemps to build it.

This model is consistent with most of the Internet peer-to-peer systems, where a process may send messages to another one if and only if it knows its IP address. To discover identifiers, a process may receive them from another process. We abstract out the implementation details in a simple resource discovery service that provides identifiers of other processes to processes that query it, thus also giving each process an entry point in the system.

Since processes can leave the system at any time, it is necessary for the neighbors of a process to be able to decide whether it is still part of the system. Otherwise, the identifiers of crashed processes could not be removed and would prevent the system from converging. Detecting such failures in a purely asynchronous system is impossible [7], so in practice, protocols such as TCP rely on timers, assuming that the Internet is not really asynchronous. In this paper, we use theoretical devices called *failure detectors* [2] to abstract out this partial synchrony: rather than making timing assumptions, we suppose that the system provides a failure detection service.

The combination of a resource discovery service and a failure detector yields a new computation model in which the classical notion of a neighbor list does not appear. In this paper, we argue that this model allows to design highly scalable distributed algorithms, as needed in large scale systems, while retaining the ability to prove their correction formally. We illustrate this with a self-stabilizing algorithm that build a spanning tree over the aforementioned complete graph. Building such a virtual topology allows to bound the number of neighbors each process has, which is necessary because it is not practical to multiplex many communication streams in today's operating systems. A user algorithm can then be composed with ours, i.e. we provide the tree as a service.

The rest of the paper is organized as follows. We describe our model in Section 2, then discuss related work in Section 3. We give the spanning tree algorithm in Section 4, prove its correctness in Section 5 and show the results of experimental performance measurements in Section 6. We conclude in Section 7.

---

[*] This work is partially funded by the PCRI/INRIA Futurs - Project Grand-Large and ACI Grid (French incentive).

[†] A preliminary version of this work was presented as a brief announcement in SSS 2006 [13].

2 Model

Definition 1. The *state* of a process is the set of its variables and their values. The *state* of a channel is the ordered list of the messages it contains. A *configuration* is a set $\mathcal{I}$ of process identifiers, a state for each $i \in \mathcal{I}$ and a state for each channel $c_{a \rightarrow b} \forall a, b \in \mathcal{I}^2$.

Definition 2. An *execution* is an alternate sequence $C_1, A_1, \ldots, C_i, A_i, \ldots$ such that $\forall i \in \mathbb{N}^*$, applying transition A_i to configuration C_i yields configuration C_{i+1}.

Definition 3. A *suffix* of an execution $C_1, A_1, \ldots, C_i, A_i, \ldots$ for $k \in \mathbb{N}$ is the alternate sequence $C_{1+k}, A_{1+k}, \ldots, C_{i+k}, A_{i+k}, \ldots$.

Definition 4. An algorithm is self-stabilizing to $\mathcal{L}$ if and only if (correctness) every execution starting from a configuration of $\mathcal{L}$ verifies the specification, (closure) every configuration of all executions starting from a configuration of $\mathcal{L}$ is a configuration of $\mathcal{L}$ and (convergence) starting from any configuration, every execution reaches a configuration of $\mathcal{L}$.

We use self-stabilization, as defined by Dijkstra [4], to design a fault-tolerant algorithm: after faults bring the system to an arbitrary configuration, the convergence property ensures that it returns to a legitimate configuration.

We denote by $(\mathcal{I}, <)$ the totally ordered finite set of process identifiers in a system and by $P \subseteq \mathcal{I}$ the set of *correct* processes, i.e. those that do not stop (crash). The other processes are stopped in the initial configuration of any execution : this corresponds to the failures that can happen before stabilization occurs. We assume the existence of lossless unidirectional FIFO links, each having a capacity bounded by an unknown constant, between each pair of processes. Channel failures such as message loss or alteration are also captured in the arbitrary initial configuration. We address the issue of writing an algorithm as if the channels were of unbounded capacity in a system where this is not the case in the same way as Afek and Bremler [1].

2.1 Services

The oracle is a formalized version of the concept of resource discovery, as used in large scale systems. It is intended to replace the neighbor list used in classical distributed systems. A process executing a guarded rule can query it, and the answer is an identifier in $\mathcal{I}$. In order to ensure the connection of the virtual topology, the collection of all the oracles has to satisfy a global property. Formally, in any suffix of an execution, if a set S of processes query their resource discovery service an infinite number of times then each process $s \in S$ obtains the highest identifier in S at least once.

The failure detector follows the definition given by Chandra and Toueg [2]: a process can query it as part of the execution of a rule, and it returns information on the other processes in the system. This information is generally unreliable, the constraints depend on the *class* of detectors in which the device is. Such a detector serves to overcome in a simple and elegant way the impossibility of solving the consensus problem in a purely asynchronous system [7]. Its implementation was studied by Chen, Toueg and Aguilera [3]. Interestingly, Chandra and Toueg's view of their failure detectors in practice matches the self-stabilization paradigm: the system behaves according to its specification most of the time and may experience infrequent transient failures. This can be modeled, as in this paper, by initializing it arbitrarily and then assuming a failure-free run.

Our model is slightly different from that of Chandra and Toueg since we cannot afford to have a device that returns a list of potentially all the process identifiers in the system due to its large size. Therefore, our detectors provide instead a function $suspect : \mathcal{I} \rightarrow boolean$. This model is equivalent to the original one. To map Chandra and Toueg's model to ours, the $suspect$ predicate can be implemented as follows: true if the process does not belong to the suspect list, false otherwise. Reciprocally, to simulate Chandra and Toueg's model in ours, it is enough to build the list of suspects by applying the $suspect$ predicate to the whole set $\mathcal{I}$.

In this work, all the failure detectors are, according to Chandra and Toueg's nomenclature, in class $\diamond\mathcal{P}$, i.e. eventually perfect. In our model, where all runs are failure-free since all failures are captured in the initial configuration by the self-stabilization model, it is defined as follows:

Definition 5. A failure detector is in $\diamond\mathcal{P}$ if and only if after a finite number of queries, its $suspect$ function returns $true$ if and only if the given identifier is in P and this property remains true from then on.

Theorem 1. *A self-stabilizing algorithm that builds a spanning tree in an asynchronous system augmented with failure detectors requires these detectors to be in* $\diamond\mathcal{P}$.

Proof. We use Segall's Propagation of Information with Feedback (PIF) algorithm [15]. It allows any process p to obtain the list of all the processes in the prebuilt spanning tree.

In the terminology of failure detection, a detector D' is *weaker* than a detector D if and only if there exists a *reduction algorithm* that transforms D into D'. Our proof consists in taking any asynchronous distributed system augmented with a failure detector FD in which a spanning tree can be built by a self-stabilizing algorithm and show that it is possible to implement an eventually perfect failure detector from it. As a result, any failure detector in $\diamond\mathcal{P}$ is weaker than FD, which implies that we need an eventually perfect failure detector to solve the spanning tree problem.

Consider a system in which the spanning tree problem is built by an algorithm A. Let AP be the following algorithm: in an infinite loop, execute A then PIF. From AP, let us build a failure detector DAP as follows: when it is queried, DAP gives as its suspect list all the identifiers in P except itself and those it obtained from the last completed call to PIF. When A is stabilized, PIF returns exactly the list of all process in the system and this property remains true. Thus DAP belongs to class $\diamond\mathcal{P}$. $\square$

2.2 Execution

The algorithm is given as a set of guarded rules. Each guard is a boolean expression that can involve the availability of an incoming message, and each rule consumes the message (if any), then can modify the process' local state and send messages. From a realistic point of view, a distributed scheduler should be assumed, but because of the communication model, no two processes can interfere with each other, so the proof is written under a centralized scheduler. We assume that the scheduler is fair, i.e. any transition that is enabled infinitely many times is eventually triggered.

To account for process identifiers that correspond to stopped (crashed) processes or to no process at all, we adopt the convention that any message sent to a stopped process is lost and that the only entity in the system that may send a message is a correct process.

3 Related Works

To design a distributed algorithm, one needs processes and a device for them to communicate. One such device is shared memory, in which many spanning tree algorithms were designed [10]. In this model, each process can read from a memory area that belongs to certain other processes, its neighbors. This memory area can contain the whole state of the neighbor or a smaller piece of data (shared registers). While this model is useful for a small scale system, like a microprocessor, it is not appropriate for a large scale system, mainly because the performance impact of maintaining a shared memory area is very high in this context.

The other classical way to communicate between processes is message passing, where pairs of process are provided with incoming and outgoing channels that can contain messages. A process can put a message into an outgoing channel as part of the execution of its code, and it is delivered to the process that is at the other end at a later stage of the execution of the algorithm. Dolev, Israeli and Moran [6] studied the differences between the two models in the context of self-stabilization. This model is adapted for geographically distributed systems made of computers linked by a network such as the Internet since the channels work in the same way as network-based communications.

However, algorithms written in message passing environments usually require all the processes to have access to a complete and up to date list of their neighbors. In a fully connected large scale system, such as a peer to peer system, where this list can be made of hundreds of thousands of processes, this approach is not realistic because of the amount of memory required to store the list and of network traffic required to keep it up to date.

Afek and Bremler [1] and Gupta and Srimani [11] used such a message passing model, where each process has to know the list of all its outgoing links and the underlying topology is not equivalent to a complete graph, to build spanning tree in a self-stabilizing fashion. Garg and Agarwal [9] solve the same problem assuming the processes are numbered sequentially, which is not the case in practice. We choose the strictly weaker and more realistic hypothesis of only requiring a total order on the process identifiers and avoid storing a neighbor list for scalability.

Existing peer to peer overlays do need to build trees in a way that is essentially self-stabilizing. The actual structures that are built are usually tailored for their purposes, e.g. a distributed hash table [8] or a Plaxton tree [14]. This the main reason that led us to chose this problem for demonstrating our model, since it highlights both its appropriateness for large scale systems and the ability to compose algorithms offered by self-stabilization [5].

4 Spanning Tree Algorithm

4.1 Presentation

In this section, we present the self-stabilizing spanning tree algorithm. The tree is distributed among all the processes and described by their $parent$ and $children$ fields.

We keep the topology free of cycles by means of a *global invariant*: the identifier of a process must be lower than that of its parent and greater than these of its children. In graph theory, this is known as the *heap invariant*.

Roughly speaking, every process is responsible for checking the consistency of its *neighborhood*, i.e. its parent and its children, using its failure detector to eliminate stopped processes, making sure its parent considers it as a child and vice versa.

In addition, every process that is its own parent, i.e. is root, is responsible for connecting to new processes via the resource discovery service that provides it with identifiers. The root r only sends to the new process p a connection request message ($Exists$) if $p > r$ in order to enforce the global invariant.

The complete algorithm is given in Subsection 4.2. Each process has two fields (local variables): $parent$, to store the identifier of its parent in the tree (or its own identifier for a root), and $children$, where it writes the set of its children.

The global invariant is enforced by the $Sanity\_check$ procedure. Line SC_2 simply ensures that $children_p$ sets initialized with too many components are reset. The other procedure, $Detect\_failures$, checks that the processes in the neighborhood of p are still members of the system. This is the only point where a process sends a query to the failure detection service, thus a process will eventually check the availability of its neighbors only. Both procedures are called at the beginning of each guarded rule. This makes sure that all the rules are executed in a clean environment.

There are five guarded rules in the algorithm. The first one is guarded by *true*, which means in practice that it is called regularly by each process. It performs the verifications described above in the neighborhood of the process.

The purpose of the rules that react to $Neighbor?$ and $NotNeighbor$ is to maintain the consistency of the process neighborhood. The $Neighbor?$ message is sent spontaneously, and silently ignored by the receiver if it is in the sender's neighborhood. If it is not, the receiver attemps to add the sender to its neighborhood. If this is impossible, it sends back a negative acknowledgement ($NotNeighbor$) that causes the sender to delete the receiver from its neighborhood.

The rules that handle $Exists$ and $YouAreMyChild$ messages control the merging of trees. Informally, a process p sends $Exists(id_p)$ to q in order to ask q to adopt p.

A process q that receives $Exists(id_p)$ thus checks whether it should do so (in particular w.r.t. the global invariant), and whether it can (this requires having less than δ children). If q should adopt p but cannot, a finer analysis decides whether q drops a child in favor of p or forwards the $Exists(p)$ message to one of its own children. In any case, q makes sure that p's request is eventually satisfied.

When a process p adds q to its set of children, it sends $YouAreMyChild(p)$ to q. Upon receiving this message, q checks whether it should accept q as its parent and does so if and only if it does not break the global invariant and q is root. This last condition reduces the number of topology changes during the convergence period.

When the system is stabilized, only the process with the highest identifier is a root and there is a single tree. Every process communicates only with its neighbors and the only messages transmitted between two processes are $Neighbor?$ requests. Moreover, only the root continues to query the resource discovery service. If the root receives an identifier, it is lower than its own if it is part of the system, so $Exists(id_{root})$ messages do not circulate. Every process checks, through the failure detection service, the availability of its neighborhood only.

4.2 Algorithm

Constants

- $id_p : id$ identifier of process p.
- $\delta : \mathbb{N}^*$ bound on the degree.

Variables (per process p)

- $parent_p : id$ p's parent.
- $children_p : set\ of\ id$ p's children.

Messages

- $Exists(id)$: sent by root processes to contact new processes for merging subtrees.
- $YouAreMyChild(id)$: sent by processes to accept a new process as its child.
- $Neighbor?(id)$: sent by all processes to check consistency of their neighbors.
- $NotNeighbor(id)$: negative acknowledge of $Neighbor?$.

Procedures and functions

- $RD\_Get()$: id returns an identifier according to the specification of the resource discovery service.
- $suspect(id_p : id) : boolean$ returns true if and only if p is suspect according to the failure detection service.
- $Neighborhood(p : process) : set\ of\ id = (children_p \cup \{parent_p\}) \setminus \{id_p\}$
- $Sanity\_check(p : process) : Void =$

```
$SC_1$ if $parent_p < id_p$ then $parent_p \leftarrow id_p$
$SC_2$ if $|children_p| > \delta$ then $children_p \leftarrow \emptyset$
$SC_3$ $children_p \leftarrow \{id_q \in children_p | id_q < id_p\}$
```

- $Detect\_failures(p : process) : Void =$

```
$DF_1$ if $suspect(parent_p)$ then $parent_p \leftarrow id_p$
$DF_2$ $\forall\ id_q \in$ $children_p$: if $suspect(id_q)$
            then $children_p \leftarrow children_p \setminus \{id_q\}$
```

Guarded rules

```
**true** $\rightarrow$
$T_1$ $Sanity\_check(p)$
$T_2$ $Detect\_failures(p)$
$T_3$ $\forall\, id_q \in Neighborhood(p)$
          **send** $Neighbor?(id_p)$ **to** $q$
$T_4$ **if** $parent_p = id_p$
$T_5$ **then** $let\ id_q = RD\_Get()$
$T_6$       **if** $id_q > id_p$ **then send** $Exists(id_p)$ **to** $q$
```

```
**Reception of** $Neighbor?(id_q)$ $\rightarrow$
$N?_1$ $Sanity\_check(p)$
$N?_2$ **if** $id_p < id_q$ **then**
$N?_3$   **if** $parent_p = id_p$ **then** $parent_p \leftarrow id_q$
$N?_4$ **else if** $id_q \notin children_p$ **then**
$N?_5$   **if** $(|children_p| < \delta) \vee (|children_p| = \delta \wedge$
           $\exists id_r \in children_p s.t.\ id_r < id_q)$ **then**
$N?_6$     $children_p \leftarrow (children_p \setminus \{id_r\}) \cup \{id_q\}$
$N?_7$   **else if** $id_p \neq id_q$ **then**
$N?_8$   **send** $NotNeighbor(id_p)$ **to** $q$
```

```
**Reception of** $Exists(id_q)$ $\rightarrow$
$E_1$ $Sanity\_check(p)$
$E_2$ **if** $id_q < id_p \wedge id_q \notin children_p$ **then**
$E_3$   **if** $|children_p| < \delta$ **then**
$E_4$     $children_p \leftarrow children_p \cup \{id_q\}$
$E_5$     **send** $YouAreMyChild(id_p)$ **to** $q$
$E_6$   **else if** $\{id_r \in children_p | id_r > id_q\} \neq \emptyset$
$E_7$       **then** $let\ id_s \in \{id_r \in children_p s.t.\ id_r > id_q\}$,
             **send** $Exists(id_q)$ **to** $s$
$E_8$       **else** $let\ id_s \in children_p$,
$E_9$           $children_p \leftarrow (children_p \setminus \{id_s\}) \cup \{id_q\}$
$E_{10}$          **send** $YouAreMyChild(id_p)$ **to** $q$
```

```
**Reception of** $NotNeighbor(id_q)$ $\rightarrow$
$\neg N_1$ $Sanity\_check(p)$
$\neg N_2$ **if** $parent_p = id_q$ **then** $parent_p \leftarrow id_p$
$\neg N_3$ $children_p \leftarrow children_p \setminus \{id_q\}$
```

```
**Reception of** $YouAreMyChild(id_q)$ $\rightarrow$
$Y_1$ $Sanity\_check(p)$
$Y_2$ **if** $parent_p = id_p \wedge id_q > id_p$ **then** $parent_p \leftarrow id_q$
```

5 Stabilization of the Algorithm

Definition 6 ($\mathcal{L}$)**.** Let Max be the process with the highest identifier in system S, P the set of processes of S and $\{c_{p \rightarrow q}, \forall p, q \in P s.t. p \neq q\}$ the set of communication channels. Since the set of processes does not change during the executions we consider for the purpose of proving the algorithm, we refer to the processes as $\rho_0 \dots \rho_{|P|-1}$ where $\rho_0 = Max$ and $\forall i \in [1..|P|-1, \rho_i$ is the process with the highest identifier in $P \setminus \{\rho_0 \dots \rho_{i-1}\}$. A configuration C is in $\mathcal{L}$ if and only if, in C, $\forall p \in P$,

$$\begin{cases} p \neq Max \Rightarrow \exists p_1, p_2 \dots p_n \in P \text{ s.t.} & \\ \quad (p = p_1) \wedge (p_n = Max) & \\ \quad \bigwedge_{i=1}^{n-1} (parent_{p_i} = id_{p_{i+1}} \wedge id_{p_i} \in children_{p_{i+1}} & (1) \\ parent_p \geq id_p & (2) \\ children_p = \{q \in P \text{ s.t. } parent_q = id_p\} & (3) \\ |children_p| \leq \delta & (4) \end{cases}$$

Additionally, $\forall p, q \in P s.t. p \neq q, c_{p \rightarrow q}$ may either be empty or, if $q \in neighborhood(p)$, contain any number of $Neighbor?(p)$ messages (5).

Condition (1) implies that there exists a unique path from any process to Max. Conditions (1) and (2) imply that Max is the only root and that any legitimate configuration satisfies the global invariant. Condition (3) ensures that any process but Max is a child of another process in the system and that only processes of the system are in the tree. Condition (5) implies that no message will break the spanning tree.

Lemma 2 (Closure)**.** *Let* $E = C_1, A_1, \dots, C_i, A_i, \dots$ *be an execution of the system. If* $C_1 \in \mathcal{L}$ *then* $\forall i \in \mathbb{N}^*$, $C_i \in \mathcal{L}$.

Proof. Let us consider the set of legal actions for a process p:

- Guard *true*.

  T_1: Sanity_check: the condition in SC_1 is false because of Condition (2) of the definition of $\mathcal{L}$and that of SC_2 is false because of Condition (4). If the condition in SC_3 was true for $id_q \in children_p$ then, by Condition (3), this would mean that $parent_q = id_p$ and thus $parent_q < id_q$, which would break Condition (2). Therefore, nothing happens.

  T_2: Detect_failures: the condition in DF_1 is false because Condition (3) of the definition of $\mathcal{L}$implies all children of p are in P. The condition in DF_2 is false because if $p \neq Max$, Condition (1) of the definition of $\mathcal{L}$implies that the parent of p is in P, and if $p = Max$, Condition (2) implies the parent of Max is Max, and thus is in P.

  T_3: $Neighbor?(p)$ messages are sent in communication channels $c_{p \rightarrow q}$ for all $q \in neighborhood(p)$, which matches the condition on messages in $\mathcal{L}$.

  T_4: because of Conditions (1) and (2), the condition is only true for Max. However, by definition of Max, there is no $q \in P$ s.t. $id_q > id_{Max}$. Therefore, Max does not send a message to a process in P. It can send an $Exists$ message to a stopped process, but then it is lost and thus does not contradict the definition.

- Reception of a $Neighbor?$ message: for Sanity_check ($N?_1$) see above. The condition in line $N?_2$ cannot be true because of the condition on messages in a legitimate configuration: p can only receive a $Neighbor(q)$ message from a process $q \in Neighborhood(p)$. Notice that consuming the Neighbor? message could not make the resulting configuration illegitimate since it does not break the condition on messages in $\mathcal{L}$.

- All other guards are closed by definition of $\mathcal{L}$. ☐

Lemma 3 (Correction)**.** *Let E be an execution of the system starting in configuration C. If $C \in \mathcal{L}$ then E verifies the specification.*

Proof. First, notice that because of closure (Lemma 2), it is enough to prove that any legitimate configuration is correct with respect to the specification.

Conditions (1) and (2) of the definition of $\mathcal{L}$ imply the existence of a unique root, namely Max. Condition (1) implies the existence of a unique path from any node to the root. This means that in any legitimate configuration, the topology described by the processes is a tree. By Condition (4), the degree of this tree is bounded by δ. ☐

The proof of convergence is done in three steps. Firstly, we prove that every execution of the system eventually reaches a configuration from which a few basic properties remain true throughout the execution. Secondly, we define a notion of stable process that formalizes the fact of irrevocably choosing a parent. We prove that a stable process remains so and that every process eventually becomes stable. Thirdly, we prove that a system in which all the processes are stable eventually reaches a legitimate configuration.

Definition 7 (Consistent configuration)**.** Let C be a configuration of system S and p be a process of S, the state of p is consistent in C iff $|children_p| \leq \delta \wedge parent_p \geq id_p \wedge \forall id_q \in children_p, id_q < id_p \wedge \neg suspect(parent_p) \wedge c \in children_p \Rightarrow \neg suspect(c)$.

Similarly, a message in a channel is consistent if it results from the complete application of a guarded rule.

Remark 1. Since the global invariant holds for all processes, a consistent configuration does not contain any cycle, i.e. any set of processes $P_1 \ldots P_n$ where $\forall i \in [1..n-1], parent_{P_{i+1}} = P_i \wedge parent_{P_1} = P_n$.

It is straightforward to see that from any initial configuration, the system eventually reaches a consistent configuration. Inconsistent states are handled by procedure $Sanity\_Check$, as already discussed. Inconsistent messages eventually reach their destination because all channels are FIFO and the scheduler is fair. The only message that can be forwarded is $Exists$, but only to a child whose identifier is higher than the parameter of the message, thus it can only happen a bounded number of times.

For the purpose of proving the algorithm, we consider an execution starting in a consistent configuration, in which all the failure detectors are converged and no process ever stops (crashes), therefore $suspect(id_p)$ is true if and only if p does not belong to the system. The spontaneous rule calls $Detect\_Failures$, which eliminates such processes, and new identifiers are only written in a process field upon reception of $Exists(id_p)$ or $Neighbor?(id_p)$, where the message was originally sent by p, which is thus alive. Therefore, the $Detect\_Failures$ procedure has no effect during the executions shown below and is not considered. Moreover, the $Sanity\_Check$ procedure does not alter the state of a consistent process. Thus, in what follows, we do not consider the execution of this procedure.

Definition 8 ($Stable(C)$)**.** Let P be the set of processes of system S, $V = \{c_{p \rightarrow q}, (p,q) \in P^2\}$ the set of communication channels between any couple p, q of P^2, and let Max denote the process with the highest identifier. Let C be a configuration of S. The stable processes of C, denoted by $Stable(C)$, is the set of processes such that any p in $Stable(C)$ is in P and verifies:

$$\left\{ \begin{array}{ll} \forall q \in P, id_q > id_p \Rightarrow q \in Stable(C) & (S_{rec}) \\ p \neq Max \Rightarrow \exists p_1, p_2 \ldots p_n \in P \text{ s.t.} & \\ \quad (p = p_1) \wedge (p_n = Max) & \\ \quad \bigwedge_{i=1}^{n-1} (parent_{p_i} = id_{p_{i+1}}) & \\ \quad \bigwedge_{i=1}^{n-1} (id_{p_i} \in children_{p_{i+1}}) & (S_{path}) \\ parent_p \geq id_p & (S_{parent}) \\ \forall q_1, q_2 \in P, Exists(p) \notin c_{q_1 \rightarrow q_2} & (S_e) \\ NotNeighbor(id_p) \notin c_{p \rightarrow parent_p} \wedge & \\ \quad NotNeighbor(id_{parent_p}) \notin c_{parent_p \rightarrow p} & (S_{nn}) \end{array} \right.$$

Roughly speaking, in a given configuration, the *stable* processes will not change their parents in the rest of the execution and are connected to the "final" tree.

Condition S_{rec} implies that p is stable only if processes with higher identifiers are also stable. This leads to a progression of the Stable set from the highest identifiers to the lowest. The S_{path} condition ensures that stable processes are part of the tree and, in conjunction with the S_{parent} condition, that the path to the root is made of stable processes. S_{path} also ensures that the parent of p acknowledges the stability of p, since its parent knows that p is its child and will remain so (since there is no $NotNeighbor$ message between p and its parent according to S_{nn}). Condition S_e also implies that no previous connection request ($Exists$ messages) from a stable process remains in the system. S_e and S_n ensure that no process rejects a child.

Theorem 4. *Let $C_0, A_0, C_1, A_1 \ldots$ be an execution of the system. Then $Stable(C_0) \subseteq Stable(C_1)$.*

Proof. By induction. Let us first consider a configuration C_0 s.t. $Stable(C_0) = \{Max\}$ and show that Max remains stable in C_1. S_{rec} and S_{path} are true for Max.

S_e is true because the only place in the algorithm where an $Exists$ message is sent out is line T_6, where this is done only to a strictly greater process. S_{nn} is true as well because the only place in the algorithm where a process sends out a $NotNeighbor$ message is sent out is line $N?_8$, where it does not send it to itself. S_{parent} is not broken because in all the places in the algorithm where a value different from id_p is written in $parent_p$, namely $N?_3$ and Y_2, this value cannot be lower than id_p.

We now consider a configuration C_0 such that $Stable(C_0) = \rho_0, \ldots, \rho_k$. Our induction hypothesis is: $\rho_0 = Max \ldots \rho_{k-1} \in Stable(C_1)$. We now show that $\rho_k \in Stable(C_1)$.

Since all the higher processes remain stable, S_{rec} still holds. For the same reason, there are only two ways of breaking S_{path}: either p changes the value of $parent_p$, or $parent_p$ deletes p from its set of children.

The former requires the execution by p of one of the following lines: 1) $N?_3$ and Y_2 are not executed because $parent_p \neq id_p$. 2) $\neg N_2$ is not executed because there was no $NotNeighbor$ message in $c_{parent_p \to p}$.

The latter requires the execution by $parent_p$ of one of the following lines: 1) $N?_4$ is not executed because $parent_p$ is stable and thus verifies condition $S_{N?}$: if it receives $Neighbor?$ from a stable process q, then $q \in Neighborhood(p)$. 2) $\neg N_3$ is not executed because there was no $NotNeighbor$ message in $c_{p \to parent_p}$. 3) E_9 is not executed because this requires receiving a message $Exists(id_q)$ s.t. $q > p$, but then q is stable and thus verifies S_e, i.e. there is no such message.

S_{parent} cannot be broken because this would require that p change the value in $parent_p$, which is proven impossible above. S_e is not broken because the only places in the algorithm where an $Exists$ message is sent out are line T_6, where this is done only by roots, and E_7, where a message $Exists(id_q), \forall q$ can only exist in C_1 if there was already such a message in C_0, which is not the case here. S_{nn} is not broken because this would require the execution of line $N?_8$ by either p or $parent_p$, but this is impossible because p and $parent_p$ are in $Stable(C_0)$, moreover $parent_p \in Neighborhood(p)$ and $p \in Neighborhood(parent_p)$.

We conclude that $Stable(C_0) \subseteq Stable(C_1)$. ☐

We now prove that the set of stable processes eventually grows until all processes are stable. In what follows, C is a configuration where there is at least one non-stable process and m is the highest such process in C.

Lemma 5. *In an execution starting with C, no process $s \in Stable(C)$ s.t. $|children_s \cap Stable(C)| < \delta$ can send $NotNeighbor(id_s)$ to m.*

Proof. The only place in the algorithm where $NotNeighbor$ messages are produced is upon reception of $Neighbor?$.

Let s be a process that receives a $Neighbor?$ message. Since $id_s > id_m$ by definition of $Stable$ and m, s takes m as a child because it has at least one child lower than id_m and does not produce a $NotNeighbor$ message. ☐

Lemma 6. *There exists a stable process p such that $|children_p \cap Stable(C)| < \delta$ such that m irrevocably writes id_p in its field $parent_m$ and p irrevocably writes id_m in its field $children_p$.*

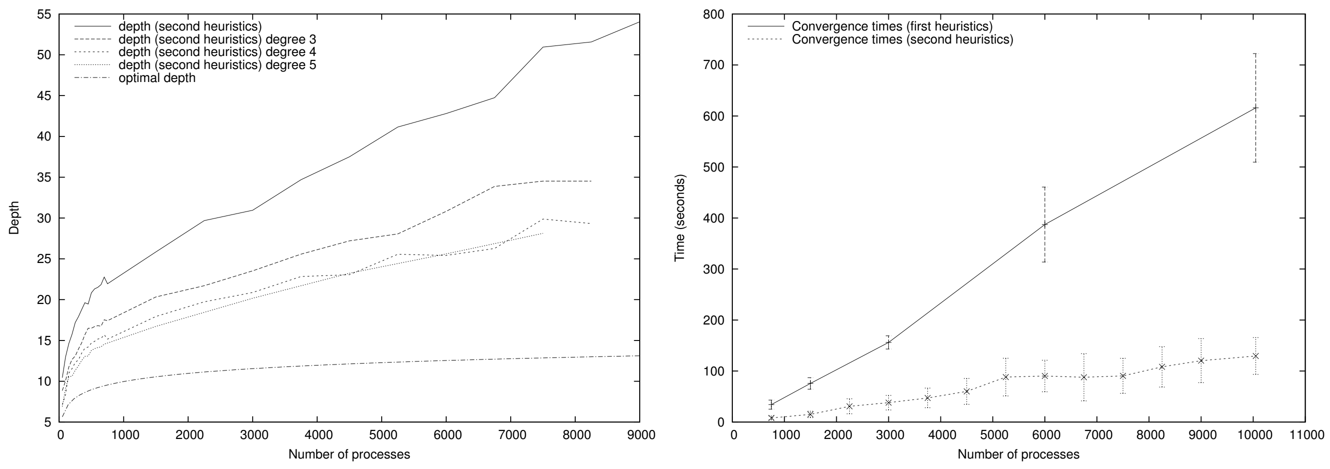
Proof. Suppose $parent_m \notin Stable(C)$. Then, by definition of m, $parent_m < id_m$, and thus the next execution of $Sanity\_Check$ will reset $parent_m$ to id_m. This eventually happens because of the spontaneous rule. Subsequently, m will only accept a parent greater than itself: the check is performed in each place in the algorithm where a write operation is performed on $parent_m$.

Suppose $parent_m = id_m$, i.e. m is root. Then m satisfies the following properties: 1) m executes its spontaneous transition an infinite number of times. This is true by hypothesis on the scheduler. 2) As part of the spontaneous transition, m queries its oracle (line T_5). Since it does so an infinite number of times, m gets the address of at least one process h, higher than itself ($id_h > id_m$) by definition of the oracle. 3) m sends an $Exists(id_m)$ message to h (line T_6).

Let us now turn to r, the receiver of one of the $Exists(id_m)$ messages that are sent out by m. There are three cases: 1) r has less than δ children. It then adds m to its set of children and sends $YouAreMyChild(id_r)$ to it (line E_4). 2) r has δ children and none of them is greater than m. It then also adds m to its set of children and sends $YouAreMyChild(id_r)$ to it (line E_9). 3) r has δ children and at least one of them, t, is greater than m. This implies that t is stable. Then r forwards the $Exists(id_m)$ message to t (line E_7).

Since the forwarding only takes place downwards in the tree and among stable process, it can only occur a finite number of times. Thus, eventually a stable process u writes id_m in its set of children and sends $YouAreMyChild(id_u)$ to it. Upon reception of this message, since $id_u > id_m$, if $parent_m$ is still id_m then m sets $parent_m$ to id_u (line Y_2).

Now suppose $parent_m \in Stable(C)$ and let us examine the possibilities for m to write another value into this field. There are three places in the algorithm where such a write operation takes place: 1) In the $Sanity\_Check$ procedure, $parent_m$ is erased if it is lower than id_m, which is not the case here. 2) Upon reception of $YouAreMyChild$, a write operation can only take place if $parent_m = id_m$, which is not the case here. 3) Upon reception of $NotNeighbor(id_p)$, if $id_p = parent_m$ then $parent_m$ is reset to id_m (line $\neg N_2$). However, by Lemma 5, no such message is produced. Therefore, this case cannot happen.



(a) Depth of the spanning trees.

(b) Convergence time

Figure 1. Scalability of the spanning tree algorithm.

Since $parent_m = id_p$ holds for the rest of the execution, $id_m \in children_p$ immediately follows. $\square$

Corollary 7. *m eventually becomes stable.*

Proof. m already satisfies conditions S_{rec}, S_{path}, S_{parent} and S_{nn}. The only line in the algorithm where m could send $Exists(id_m)$ is T_6, but it does not do so because $parent_m \neq id_m$. It is thus enough to show that the remaining $Exists(id_m)$ messages are consumed.

Upon reception of an $Exist(id_m)$ message, a process that is lower than m, i.e. any unstable process, ignores it (line E_2). It it however possible that a stable process s s.t. $id_s \neq parent_m$ add id_m to its set of children and send $YouAreMyChild(id_s)$ to it. Then the following properties apply: 1) Since $parent_m$ is now permanently set to another process (Lemma 6), m ignores this message (line Y_2). 2) s eventually executes its spontaneous rule (by hypothesis on the scheduler) and thus sends $Neighbor?$ to m (line T_3). 3) Upon reception of $Neighbor?(id_s)$, $s \notin Neighborhood(m)$. This is because (a) $id_s > id_m$ and thus $id_s \notin children_m$ and (b) $id_s \neq parent_m$. Therefore, m replies by sending $NotNeighbor(id_m)$ to s (line $N?_8$). 4) Upon reception of $NotNeighbor(id_m)$, s removes id_m from its set of children (line $\neg N_3$). $\square$

Toward a legitimate configuration

Lemma 8. *Let E be an execution of the system S starting from a configuration C s.t. $\forall p \in P$, the state of p is consistent and $p \in Stable(C)$. There exists a configuration L of E such that $L \in \mathcal{L}$.*

Proof. S_{path} is the same as condition (1) of the definition of $\mathcal{L}$ and S_{parent} is the same as condition (2). Condition (4) is satisfied because all processes are always in a consistent state during E by assumption.

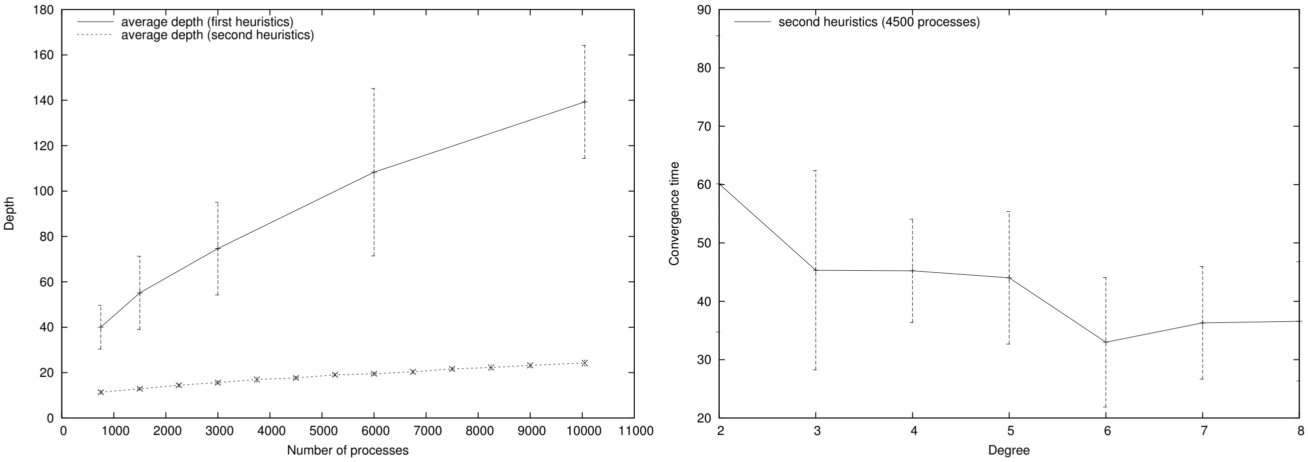
If in C, condition (3) is not satisfied, then $\exists p, q \in P$ s.t. $id_p \in children_q \wedge id_q \neq parent_p$. If $p \notin P$, process q eventually rejects p from its children by executing DF_2. If $p \in P$, process q eventually rejects p from its children: q eventually sends $Neighbor?(id_q)$ to p, and p eventually answers $NotNeighbor(id_p)$ to q (because $id_q \notin children_p$ since q and p are in a consistent state and it is impossible that $id_p < id_q \wedge id_q < id_p$), and then q rejects p from its children. Thus, every process eventually satisfies condition (3). Moreover, if a process satisfies condition (3) in some configuration of E, then this process satisfies this condition in all subsequent configuration. Indeed, a process p can add a process q to its children iff p receives a message $Exists(id_q)$. But this cannot happen because of condition S_e.

Thus there exists a suffix E' of E s.t. in all configurations of E', conditions (1), (2), (3) and (4) are satisfied.

Conditions S_{nn} and (3) implies that in all configurations of E', we have $\forall p, q \in P$ s.t. $p \neq q, Notneighbor(id_p) \notin c_{p \rightarrow q}$. Moreover, condition S_e implies that in all configurations of E', $\forall p, q, r \in P$ s.t. $p \neq q, Exists(id_r) \notin c_{p \rightarrow q}$, and so there exists a suffix E'' of E' in which: $\forall p, q, r \in P$ s.t. $p \neq q, YouAreMyChild(id_r) \notin c_{p \rightarrow q}$. Thus, condition (5) of the definition of $\mathcal{L}$ is always satisfied in E''. $\square$

6 Experimental Measurements

We measured the performances of a simple implementation of our algorithm on an experimental cluster platform. It consists in 150 bi-Opteron machines linked by Gigabit



(a) Average depth of the nodes.

(b) Influence of the degree.

Figure 2. Other characteristics of the spanning tree algorithm.

Ethernet adapters, part of the Grid Explorer platform. This high-performance cluster allows us to run large scale experiments in a reproducible way.

To measure the scalability of the algorithm, we start it each time from a totally disconnected configuration. This is the worst case: as soon as some processes start earlier than others the convergence time is shorter. We compare two heuristics in the only place in the algorithm left up to the user : the choice of the child that is deleted (line E_9) or to which $Exists$ messages are forwarded (line E_7). The first one consists in always selecting the highest identifier, the second one in randomly choosing an eligible child following a homogeneous distribution.

6.1 Experimental Method

Measuring the convergence time of self-stabilizing algorithms is not a straightforward process. First, since any difference in the starting times of processes changes the results significantly, we synchronized them using a broadcast mechanism. The second problem is that no single node has a complete view of a whole configuration. Therefore, we added to our nodes a logging mechanism to record every modification of the state and let each experiment run until the user interrupted it. Then each process dumped its local history and a post-mortem analysis was conducted.

To obtain the times presented in the figures, we extract the first configuration (according to logical clocks) where convergence is attained, and return the time measured by the processes from the beginning of the experiment to that point according to their local clocks. If the predicate holds in no configuration, the experiment is run again during a longer time. Once an upper bound is determined, we run the experiment 20 times to obtain the mean and standard deviation that are used to plot the curves.

In order to simulate the presence of a high number of machines, we ran several instances of the program on each physical machine. We verified experimentally that this did not saturate the available CPU and network resources.

6.2 Algorithm

Our implementation is mostly straightforward. The only part that requires explanations is the timeout-triggered spontaneous rule. The heuristic we use to dynamically adapt the duration of the timeout to the activity of the system takes five time arguments: *initial*, *minimum*, *maximum*, *increment* and *decrement*. At startup, the process triggers the spontaneous rule with a period of *initial* time units. Every time the process changes its state, it subtracts *decrement* from its current timeout, with a lower limit of *minimum*. Every time the application of the spontaneous rule does not change its state, it adds *increment* to its timeout value up to *maximum*. Since the algorithm induces state modifications only when the system has not converged, this heuristic reduces the time lost waiting for a message from a process during the convergence phase and lowers the amount of processor and network usage when convergence is achieved.

We designed our resource discovery service to be efficient on a cluster, since this is the experimental testbed we used. The local daemons that provide identifiers to the processes maintain lists of identifiers. The daemons regularly communicate via multicast channels to update their lists consistently. This includes sorting them in LRU order, so that processes that stop querying their oracle are no more considered.

6.3 Results

We present in figure 1 the convergence time and the depth of the tree for 750 to 10050 processes. This figure shows that the convergence phase is divided into two stages: at first the main operation is the insertion of a process in a tree, at this point its depth is optimal, i.e. logarithmic in the number of processes. This is made more efficient by increasing the degree and thus giving each process more children slots. When the main operation becomes tree merging, the depth begins to progress linearly with the number of tree merging, that is linear in the number of nodes. Figure 1(b) shows that the second heuristics yields better performances than the first one. We explain this result below.

Figure 2 displays other characteristics of the algorithm: figure 2(a) reflects the average depth of nodes, an indicator of the quality of the trees that shows that the trees built using the second heuristics have a higher filling rate, due to the random choice for descending $Exists$ messages. This is why the second heuristics gives a better convergence time.

Figure 2(b) shows the influence of δ on the convergence time. As expected, one can see that a higher number of children slots allows the logarithmic phase to last longer, thus improving the performances.

7 Conclusion and Future Works

In this work, we propose a new model in which to design distributed algorithms for large scale systems. We replace the classical notion of a neighbor list with the combination of two devices, a resource discovery service and a failure detector, in order to avoid making unnecessary assumptions and to improve scalability.

We illustrate our model with a self-stabilizing algorithm that builds a spanning tree whose degree is bounded by δ using only $\delta + 1$ process identifiers. We present a formal proof of convergence and performance measurements of a prototype implementation of this algorithm and its services for clusters. The experimental results show that the algorithm performs well enough to argue in favor of the application of self-stabilization in practice.

Our intended followup on this work is to design other protocols, in the same model, so as to explore its viability and efficiency for different problems.

In particular, we will study other topologies suitable for large scale systems. It would also be interesting to try to define the notion of stabilisation time in this model. This would require stronger assumptions on the resource discovery service, but which ones exactly is an open question.

References

[1] Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 4(3):1–48, 1998.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, March 1996.

[3] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51, May 2002.

[4] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11):643–644, 1974.

[5] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[6] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self stabilizing message driven protocols. In *PODC91 Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 281–293, 1991.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[8] M. Freedman and D. Mazieres. Sloppy hashing and self-organizing clusters. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

[9] V. K. Garg and A. Agarwal. Self-stabilizing spanning tree algorithm with a new design methodology. Technical Report TR-PDS-2004-001, 2004.

[10] F. C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report IC/2003/38, EPFL, Technical Reports in Computer and Communication Sciences, 2003.

[11] S. K. S. Gupta and P. K. Srimani. Self-stabilizing multicast protocols for ad hoc networks. *J. Parallel Distrib. Comput.*, 63(1):87–96, 2003.

[12] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. Self-stabilizing spanning tree algorithm for large scale systems. Technical Report 1457, LRI, 2006.

[13] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. Self-stabilizing spanning tree algorithm for large scale systems (brief announcement). In A. K. Datta and M. Gradinariu, editors, *Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Lecture Notes in Computer Science, 2006.

[14] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM Press.

[15] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.