RAPPORT DE RECHERCHE

L R I

SELF-STABILISING OVERLAY NETWORK FOR EFFICIENT PROCESSES NUMBERING IN LARGE SCALE SYSTEMS

PERES O / HERAULT T

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

02/2007

Rapport de Recherche N° 1466

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Self-stabilising Overlay Network for Efficient Processes Numbering in Large Scale Systems

Olivier Peres and Thomas Herault

Univ Paris Sud; LRI UMR8623; CNRS ; INRIA; Orsay F-91405

Abstract. We introduce a self-stabilising algorithm that builds an overlay network in any large scale system, such as a peer to peer system. This allows several applications, such as efficient routing and consecutive processes numbering. 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. The algorithm uses a neighbourless model for the sake of scalability and relies on the composition of several subalgorithms.

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-stabilising algorithms are designed for distributed systems defined by their topology. Each process has a finite set of communication links to exchange messages with its neighbours. In our model, we replace the existence of a complete topology with the notion of neighbourhood, 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 [11] 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.

Using this model, we introduce a self-stabilising algorithm that builds a virtual topology over such a system. Our goal is to allow efficient operation over this topology, such as routing, and to facilitate building other topologies on top of it, such as distributed hash tables. Our algorithm benefits from the neighbourless model in two ways: it provides scalability and the ability to give a formal proof.

The global algorithm is made of three composed algorithms. The first one makes packs of processes where the cardinal of each pack is a power of two. The second algorithm links the packs to form a global structure, and the third efficiently assigns consecutive integer identifiers to the processes, a prerequisite for many higher-level algorithms.

Since processes can leave the system at any time, the neighbours of a process have to be able to decide whether it is still part of the system. Otherwise, the identifiers of crashed processes could not be removed, preventing the system from converging. Since detecting such failures in a purely asynchronous system is

impossible [7], 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 rest of the paper is organised as follows. We discuss related works in section 2, introduce our model in section 3, and present our algorithms and prove them correct in section 4. We conclude in section 5.

2 Related works

To design a distributed algorithm, one needs processes and a device for them to communicate. One such device is shared memory. Many self-stabilising algorithms to build topologies, e.g. spanning trees [10], were designed in this model where each process can read from a memory area that belongs to certain other processes, its neighbours. This memory area can contain the whole state of the neighbour 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. This consists in providing pairs of process 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-stabilisation. 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 neighbours. In a large scale 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.

Building a spanning topology, in this case a tree, was done by Garg and Agarwal [9] assuming the processes are numbered sequentially. This illustrates the practical need for such a numbering, which is normally not provided by large scale systems. It is provided by our algorithm using the strictly weaker and more realistic hypothesis of only requiring a total order on the process identifiers.

Existing peer to peer overlays do need to build spanning topologies in a way that is essentially self-stabilising. The actual structures that are built are usually tailored for their purposes, e.g. a distributed hash table [8]. Thanks to the ability to compose algorithms offered by self-stabilisation [5], the sequential numbering that we provide makes it straightforward to to build such structures in a self-stabilising way, with the added benefit of allowing to provide a formal proof of correctness.

We first illustrated the neighbourless model [11] with a spanning tree algorithm whose space complexity is $O(1)$, resulting in the scalability we seek to obtain. The average space complexity of the new algorithms we present here is also $O(1)$, thus ensuring a similar scalability. However, they are more distributed in the sense that in the general case, a large number of processes do useful work during the same physical time, which was not necessarily true in the previous algorithm. They are also more flexible, e.g. while a global leader is still elected, it is not predetermined. The numbering algorithm also shows that routing can be performed on the structure that is built in a logarithmic number of messages, i.e. as efficiently as over a balanced binary tree: this is an improvement over the arbitrary tree built by our previous algorithm.

3 Model

Our model uses the classical definitions of state, configuration and execution: 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 $c_{a\to b} \forall a, b \in \mathcal{I}^2$. 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}. A *suffix* of an execution $C_1, A_1, \ldots, C_i, A_i, \ldots$ for $k \in \mathbb{N}$ is the alternate sequence $C_{1+k}, A_{1+k}, \ldots, C_{i+k}, A_{i+k}, \ldots$. Using these notions, we can define self-stabilisation.

Definition 1. *An algorithm is self-stabilising 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-stabilisation, 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 stabilisation 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].

The oracle is a formalised version of the concept of resource discovery, as used in large scale systems. It is intended to replace the neighbour 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 all the identifiers 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-stabilisation paradigm: the system behaves according to its specification most of the time and may experience infrequent transient failures. This is modeled, as in this paper, by initialising 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 $suspect : \mathcal{I} \rightarrow boolean$. 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-stabilisation model, it means that after a finite number of queries, their *suspect* function returns *true* if and only if the given identifier is in P and this property remains true from then on.

The algorithms are given as sets 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 centralised scheduler. We assume that the scheduler is fair, i.e. any transition whose guard is evaluated to true infinitely many times is eventually drawn.

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. The contents of the channels are arbitrary in the initial configuration, though, since a self-stabilising algorithm must be able to recover from any channel failure.

4 Algorithms

4.1 Pack Algorithm

This algorithm, presented in figure 1, needs a parameter t such that $t > \log n$, where n is the number of processes in the system. It builds a maximal pack of

Variables
neighbour[0..t] : process identifiers

Definitions

- Active
  $active(0) \equiv true$
  $\forall i \in 1..t, active(i) \equiv$
  $active(i-1) \wedge neighbour[i-1] \neq \perp$
- Level
  $level \equiv \max\{i \mid active(i)\}$
- Leader
  $leader \equiv (level = 0) \vee$
  $neighbour[level - 1] < myself$

Guarded rules

- Sanity checking
  $true \rightarrow \forall i \in 0..t,$
  if suspect(neighbour[i]) $\vee \neg$ active(i)
  $\vee$ neighbour[i] = myself
  then neighbour[i] $\leftarrow \perp$
- Link maintenance
  $\forall i \in 1..t,$
  $active[i] \wedge neighbour[i] \neq \perp \rightarrow$
  send $Hello(i)$ to $neighbour[i]$
- Prospecting
  $leader \rightarrow$ let v = RD_Get() in
  if $\neg$suspect(v) $\wedge v > myself$ then
  send $Exists(level)$ to v
- Reaction to Exists
  reception of $Exists(j)$ from $v \rightarrow$
  if active(j) $\wedge\neg$suspect(v)
  $\wedge$ neighbour[j]= $\perp$
  then neighbour[j] $\leftarrow v$
- Reaction to Hello
  reception of $Hello(j)$ from $v \rightarrow$
  if neighbour[j] = $\perp \vee v >$ neighbour[j]
  then neighbour[j] $\leftarrow v$
  else if neighbour[j]$\neq v \vee \neg active(j)$
  then send $Goodbye(j)$ to v
- Reaction to Goodbye
  reception of $Goodbye(j)$ from $v \rightarrow$
  if neighbour[j] = v
  then neighbour[j] $\leftarrow \perp$

Fig. 1. Pack algorithm.

processes whose cardinal is a power of 2 and elects a leader inside it. Informally, each process p searches for a partner q at level 0. When it finds it, thus forming a pack of size 2, one of the processes becomes leader and starts looking for a partner at level 1. This partner has to be itsef the leader of a pack of size 2. Eventually, every process in the system becomes a member of exactly one pack. Each pack describes a binary tree, which makes this simple structure suitable for routing.

The resulting structure is shown on figure 2. The processes from P_1 to P_7 are represented as rectangles, each containing four neighbour variables. The pairs are denoted by solid lines, the leader of each pair is emphasised by a thicker line. So, for example, P_1 and P_3 are paired at level 1 and P_3 is the leader.

The RD_Get function is used by each process to query its oracle, it returns an identifier in $\mathcal{I}$.

Proof of self-stabilisation

We prove that our algorithm verifies the three required properties to be self-stabilising: first correctness, then closure, and lastly, convergence.

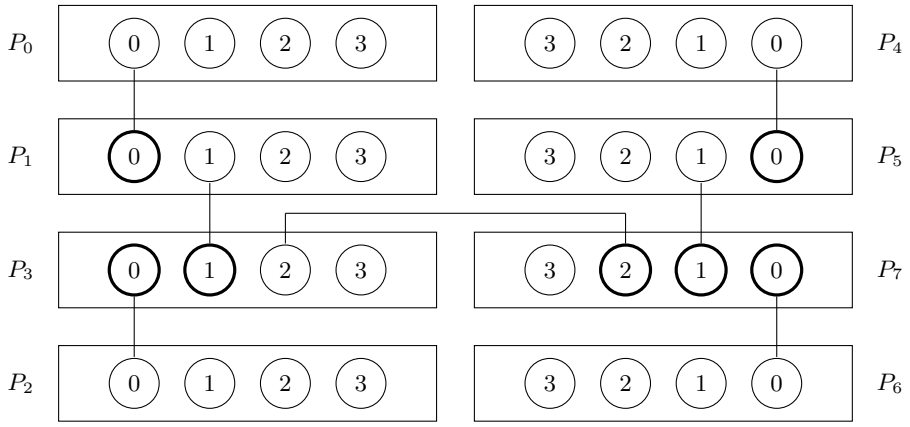

Fig. 2. Structure of a pack.

Definition 2. *Let p and q be two distinct processes. A system σ is* stable *at level l if and only if it verifies the following properties for all $m \leq l$:*

- *active(p) $\wedge$ active(q) $\wedge p \neq q \wedge$(neighbour[m](p) $= q \Rightarrow$ neighbour[m](q) $= p$).*
- *there are at most $2^{m+1} - 1$ active processes (p_i) s.t. neighbour[m]$(p_i) = \perp$.*
- *if neighbour[m](p) $= \perp$ then Exists$(m) \notin c_{q \to p}$.*
- *Hello$(m) \in c_{p \to q} \Rightarrow$ neighbour[m](p) $= q$.*
- *Goodbye$(m) \notin c_{p \to q}$.*

Definition 3. *A system is in the set $\mathcal{L}_p$ of legitimate configurations if and only if it is stable at level $\lceil \log n \rceil$.*

Lemma 1 (correctness). *Let σ be a system made of n processes in a legitimate configuration. The system comprising n processes, where n can be written as $\sum_{i \in I} 2^i$, is made of $|I|$ packs, one for each $i \in I$, of size 2^i.*

Definition 4. *A process p is* paired *at level m iff p is stable at level $m-1$ and there is a process q, stable at level $m-1$, s.t. neighbour[m]$(p) = q$. A process that is not paired at level m is* unpaired *at level m.*

Lemma 2. *A system stable at level l remains so throughout any execution.*

Proof. The possible transitions are:

- Sanity checking: no process is suspect because the failure detectors are converged, no process is its own neighbour in the initial configuration by definition of stable(l), and the possible correction affecting an inactive process does not make the configuration illegitimate since the conditions only concern active processes.

– Link maintenance and Prospecting: the only message that can be sent are $Hello$ to a neighbour, which obeys the rule on $Hello$ messages, and $Exists$ to an already paired process, which verifies the rule on $Exists$ messages.
– Reaction to Exists: by definition of stable, an $Exists(m)$ message can only be received by a process p s.t. $neighbour[m](p) \neq \perp$, thus p does nothing.
– Reaction to Hello: by definition of stable, a $Hello(m)$ message can only be sent by p to q s.t. $neighbour[m](p) = q$, thus q reacts by doing nothing.
– Reaction to Goodbye: by definition of stable, there is no such message in the channels linking stable processes together.

Remark 1. None of the above transitions can change $neighbour[m](p)$ for a process p stable at level $m \leq l$.

Corollary 1. *The set of legitimate configurations is closed under the execution of the algorithm.*

Lemma 3. *Any system σ stable at level $l-1$ and unstable at level $l \geq 0$ or unstable at level $l = 0$ eventually becomes stable at level l.*

Proof. First notice that the execution of the sanity checking rule eliminates the cases of suspect neighbours and self-connections (m, p s.t. $neighbour[m](p) = p$). Since we suppose that the failure detectors are stabilised at this point, we disregard crashed processes. Similarly, at no point in the algorithm is it possible for a process to connect to itself. Also, all the messages present in the initial configuration are consumed and all the processes have executed their sanity checking rule. Finally, the fact that σ is stable at level $l-1$ means that none of the values of *active*, *level* or *leader* can change in σ. This is because they only depend on *active* itself and $neighbour[m](p)$ for p stable at level m, and this cannot change (Remark 1).

Let z be the highest non-paired process at level l. Notice that no process can send *Goodbye* to z. Thus, if z writes the identifier of a correct process p in its *neighbour* field, then p and z become paired.

Suppose there is an execution of σ where no pair is formed at level l. Let p be a process distinct from z, not paired at level l (p has to exist, or σ would be stable at level l). As part of its spontaneous prospection rule, p sends out an infinite number of $Exists$ messages and thus, by definition of the oracle, sends $Exists$ to z. Since z has $\perp$ in its *neighbour* field, it takes p as a neighbour: contradiction.

Hence, eventually the number of process pairs at level l is maximal, which leaves at most $2^{m+1} - 1$ unpaired processes.

Theorem 1. *The pack algorithm is self-stabilising to $\mathcal{L}_p$.*

Proof. (convergence) By Lemma 3, system σ eventually becomes stable at all levels. At this point, it has reached a legitimate configuration.

4.2 List algorithm

This algorithm maintains a doubly linked list topology that connects together packs of processes created by the pack algorithm. The first process in the list is the leader of the highest-level pack, then all the other leaders appear in decreasing level order down to that of the lowest-level pack. This algorithm is to be composed with the pack algorithm. Therefore, it can read its variables and evaluate its functions, but not modify any of them.

The algorithm works as follows. Each process has a *next* and a *previous* variables that hold the identifier of the leader of the immediately lower and higher packs, respectively. The *next_level* and *previous_level* variables contain the levels of these packs. As for the pack algorithms, there are guarded rules for sanity checking, link maintenance and prospecting. Three messages are used : $ListExists$ to discover new processes, $ListHello$ to insert a process in the chain and $ListGoodBye$ to delete a process from it. The associated guarded rules insert the sender in the chain when appropriate for $ListExists$ and $ListHello$ and delete it in all cases for $ListGoodBye$.

The proof goes as follows. We first define the attractor that we use for the proof of convergence and the set of legitimate configurations.

Definition 5. *Let m be the number of packs formed by the pack algorithm. A configuration is* stable *at level $l \leq m$ if and only if the following properties hold for the l lowest pack leaders:*

- *if p is the leader of the lowest-level pack then $next(p) = \perp$ and $next\_level(p) = -1$, else $previous(p) = q$ where q is the leader of the pack whose level is immediately lower than $level(p)$ and $next\_level(p) = level(q)$.*
- *previous_level(p) = level(previous(p)), next_level(p) = level(next(p)).*
- *no channel contains a ListGoodBye message.*
- *$Hello(l) \in c_{p \to q} \Rightarrow$ previous(p) = q$\vee$ next(p) = q.*

Definition 6. *A configuration that is stable at level m and in which the leader p of the largest pack is such that $previous(p) = \perp$ and $previous\_level(p) = t$ is* legitimate. *The set of such configurations is called $\mathcal{L}_l$.*

The proofs of correction and closure are straightforward. The proof of convergence is done as follows: we now consider a system where the failure detectors are stabilised, initial messages are consumed and the pack algorithm is stabilised.

Definition 7. *Let p be a pack leader. The value of $previous(p)$ or $next(p)$ is* spurious *if the associated level does not match that of the corresponding process, i.e. $next\_level(p) \neq level(next(p))$ or $previous\_level(p) \neq level(previous(p))$.*

Lemma 4. *All spurious values are eventually eliminated.*

Proof. Since the two cases are symmetric, suppose $next\_level(p) \neq level(next(p))$. Eventually p executes its sanity checking rule, thus p sends $ListHello$ to q. If q is not a pack leader, then it replies with $ListGoodBye$, upon reception of which p writes $\perp$ in its *next* field. If q is a pack leader then $level(q) > level(p)$ by definition of p, so q also replies with $ListGoodBye$.

Since it is not possible to introduce a spurious value in the system because the level fields are updated at the same time as the *previous* and *next* fields, from now on, we suppose that no spurious value exists in the system. An immediate consequence of this is that the system eventually becomes stable at level 0.

Theorem 2. *A system stable at level l eventually becomes stable at level $l+1$.*

Proof. Let p be the leader of the smallest unstable pack and q be the leader of the largest stable pack. First, notice that if $next(p) = q$, then p cannot change the value of its *next* field. This would require one of the following:

- a process r such that $level(p) > level(r) > level(q)$ sends $ListHello$ to q, but by definition of p and q, there is no such r.
- q sends $ListGoodBye$ to p. Since q is a pack leader, this can only happen if q receives $ListHello$ from q, but in this case, since $level(p) > level(p)$ and there is no level between those two, p does not send a $ListGoodBye$ message.

Then, we prove that eventually, $next(p) = q$. Consider an execution where this is never the case. By its prospection rule, q sends $ListExists$ to all the processes returned by its oracle an infinite number of times. Thus, by definition of the oracle, q sends $ListExists$ to p. Since $level(p) > l > next\_level(p)$, p writes the identifier of q in its *next* field: contradiction.

Corollary 2. *The list algorithm is self-stabilising to $\mathcal{L}_l$.*

4.3 Zero-memory sequential process numbering

This algorithm runs on top of the other two algorithms. Its goal is to assign unique consecutive integer identifiers, starting from 0, to all the processes in the system. Using no memory, this algorithm stabilises instantly. It takes advantage of the tree structure described by a pack, as shown in figure 2, to achieve an $O(\log n)$ message complexity. Any process can send message m to process number i by sending $RouteUp(m, i)$ to itself.

First, the message is routed up to the global leader, i.e. the leader of the toplevel pack. For this, each process forwards the message to its leader. For example, in the pack described in figure 2, P_0 would send $RouteUp(m, i)$ to P_1, then P_1 to P_3, then P_3 to P_7, which is the pack leader. A pack leader forwards the message to its *previous* neighbour in the list. If there is none, the global leader is reached. It sends $RouteDown(m, i)$ to itself.

The next step is to find the pack in which the receiver is. At the list level, the processes are numbered as follows : the toplevel pack, made of 2^i processes, comprises the processes from 0 to $2^i - 1$; the next pack of 2^j processes is made of processes 2^i to $2^i + 2^j - 1$, etc. Thus, a leader of level l that receives $RouteDown(m, i)$ checks if $i < 2^l$: if so then the receiver is in this pack, else it sends $RouteDown(m, i - 2^l)$ to the next pack. If there is none, this means that $i \geq n$, where n is the number of processes in the system, and the message is simply lost.

In every pack, from the point of view of each process p of level l, the numbering is defined as follows : process 0 is p, then processes 1 to 2^{l-1} are in the subpack rooted at neighbour$[l-1]_p$, etc. In figure 2, the numbering goes as follows : $P_7 = 0$, $P_3 = 1$, $P_1 = 2$, $P_0 = 3$, $P_2 = 4$, $P_5 = 5$, $P_4 = 6$, $P_6 = 7$.

The algorithm thus forwards the message down recursively through the pack until the receiver is found. For example, if P_7 receives $RouteDown(m, 2)$, it knows that the receiver is in the pack rooted at P_3, so it sends $RouteDown(m, 1)$ to P_3, which in turn sends $RouteDown(m, 0)$ to P_1. Then P_1 delivers the message since it is the intended receiver.

4.4 Space complexity analysis

For simplicity, we present the pack algorithm using a vectorial notation for the neighbour variables, resulting in an $O(\log n)$ space complexity, where n is the number of processes in the system. In an implementation that dynamically allocates only the necessary memory, while the global leader still needs $\lfloor \log n \rfloor$ neighbour variables, the total number of process identifiers needed in the system is bounded by $2n$. Indeed, n processes have a neighbour at level 0, $\frac{n}{2}$ have a neighbour at level 1, ..., 2 at level $\lfloor \log n \rfloor$. Thus, the average number of neighbours per process is

$$
\begin{aligned}
a &= \frac{\sum_{k=0}^{k=\lfloor \log n \rfloor} \frac{n}{2^k}}{n} \\
a &= \sum_{k=0}^{k=\lfloor \log n \rfloor} \frac{1}{2^k} \\
a &= 1 + \sum_{k=1}^{k=\lfloor \log n \rfloor} \frac{1}{2^k}
\end{aligned}
$$

The upper bound of the second term is the limit of the power series:

$$
\lim_{k \to \infty} \sum_{x=1}^{x=k} \frac{1}{2^x} = 1
$$

Therefore, $a < 2$ and the average space complexity of the pack algorithm is $O(1)$. Its composition with the list algorithm, whose space complexity is $O(1)$, and the zero-memory numbering algorithm thus has an $O(1)$ average space complexity.

5 Conclusion

We introduce a set of scalable algorithm that build a virtual topology over a large scale system in which the only assumption is that any process can communicate with any other process provided it knows its identifier. This topology allows to efficiently assign consecutive integer identifiers to the processes, a prerequisite for many higher-level tasks. The average space complexity is $O(1)$, which ensures a good scalability, and the number of messages needed to route information from a node to another is $O(\log n)$.

The use of self-stabilisation allows to recover from arbitrary errors affecting variables and channels, but also to compose algorithms. To take advantage of the sequential numbering, any self-stabilising algorithm simply has to be composed with the algorithms that we present in this paper.

We intend to follow up on this work by further studying the implementation and proof of scalable algorithms in large scale systems as well as their performances, both from a theoretical and from a practical point of view.

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. T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A model for large scale self-stabilization. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2007.