On the Complexity of a Self-stabilizing Spanning Tree Algorithm for Large Scale Systems

Julien Clement*, Thomas Herault, Stephane Messika and Olivier Peres
{julien.clement,thomas.herault,stephane.messika,olivier.peres}@lri.fr
Univ Paris Sud; LRI; CNRS; INRIA; Orsay F-91405
Bat 490, Universite Paris-Sud, 91405 Orsay Cedex, France
Phone: +33 1 69 15 69 06, Fax: +33 1 69 15 65 86

Abstract

Many large scale systems, like grids and structured peer to peer systems, operate on a constrained topology. Since underlying networks do not expose the real topology to the applications, an algorithm should build and maintain a virtual topology for the application. This algorithm has to bootstrap the system and react to the arrival and departures of processes.

In a previous article, we introduced a computing model designed for scalability in which we gave a self-stabilizing algorithm that builds a spanning tree. At that time, we provided a proof of stabilization and performance measurements of a prototypal implementation. In this work, we present a probabilistic method to evaluate the theoretical performances of algorithms in this model, and provide a probabilistic analysis of the convergence time of the algorithm.

Keywords: distributed systems, self-stabilization, overlay network, time complexity, probabilistic analysis.

1. Introduction

Large scale systems are made of a large number of machines linked by a network. In the case of grids or peer to peer systems, the network is generally based on interconnected local area networks or the infrastructure of Internet service providers. Nodes communicate with each other using a network protocol, like IP, that routes the packets hop by hop.

As a result, it is necessary for large scale systems to operate without knowing the underlying topology. To achieve this, modern peer to peer systems abstract out the physical network topology by defining a *virtual* topology. Two peers are *virtual neighbors* if there is a link between them in the virtual topology. There may or may not be a physical link between them.

The normal mode of operation in this context is an *overlay*. Each peer has a unique identifier, which can be an IP address and a port or drawn from a more specialized address space, like the codomain of a cryptogaphic function. Knowing the identifier of a process is the necessary and sufficient condition to be able to send it a message. The virtual topology itself is usually optimized so that the query run fast, e.g. in a logarithmic number of hops on a Chord ring[11].

Typically, the virtual topology is built on top of the physical topology by *gossiping*. After a boostrapping phase where each peer knows a given identifier, usually a well-known peer, all of them periodically exchange their knowledge of peer identifiers and update it based on the information they receive. This behavior is modeled by *peer sampling services* [9]. They allow each node to know a small number of other nodes, ensuring that the resulting topology is connected.

This model is different from the usual model used to write distributed algorithms. Classically, each process knows the whole set of its neighbors, updated whenever a node arrives or leaves the system. It can be used directly to write distributed algorithms that can operate on large scale systems [8]. The previous works on this subject showed an algorithm that builds a spanning tree using only a constant number of process identifiers, with a formal proof of self-stabilization. Its performance was evaluated on a prototype implementation.

The experiments showed that the convergence time of the algorithm varies significantly according to the properties of the oracle. We therefore present our main contribution: the expected convergence time of the algorithm using Markov chains methods [6, 12]. We distinguish between two oracles with different behaviors: the first one relies on a uniform distribution of the identifiers and the second one follows a power law.

*Supported by Région Île de France

978-0-7695-3448-0/08 $25.00 © 2008 IEEE
DOI 10.1109/PRDC.2008.36

The rest of the paper is organized as follows. In section 2, we discuss related works. We present the computational model in section 3, the algorithm in section 4, an adapted model for the probabilistic study in section 5, and the probabilistic analysis in section 6. We conclude in section 7.

2. Related Works and Motivation

The performance of overlay networks in large scale systems are typically evaluated probabilistically. For example, the authors of Chord [11] show that the expected number of steps for lookups on their skip ring is $O(\log N)$, where N is the total number of processes in the systems. Similar studies were conducted for other overlay networks, like Pastry [10] or Tapestry [13]. The underlying assumption is that since the system is very large, the identifiers are well spread by a hash function and the running time is long, the whole system globally behaves according to the probabilistic model.

We showed [8] that it is possible to build an overlay network – a spanning tree, in this case – in a large scale system, while keeping in mind the scalability issues, and provide a formal proof of self-stabilization under a nondeterministic scheduler, as opposed to a probabilistic proof of convergence. For this purpose, we introduced a computing model designed for scalability. By avoiding to provide each node with a list of its neighbors, it allows to write algorithms that focus on local operations and make use of an oracle when necessary, to ensure a global connectivity.

Since it is proven that the algorithm eventually stabilizes in any execution, the remaining question is its theoretical performance. We therefore define a probabilistic model and compute the expected convergence time of the algorithm.

The model distributes the knowledge of the global topology in the whole system. In peer to peer systems, this is a common assumption that is typically implemented using a peer sampling service. In this context, each process has a function getPeer() that returns the identifier of another process. Using this function, the nodes get to know each other.

In order to study the behavior of an algorithm, we thus need to characterize the answers of this getPeer() function. We first take the assumption, widely accepted until recently, that we can draw process identifiers at random following a uniform distribution. Then, we switch to the more realistic view that the identifiers follow a power law, i.e. a few identifiers are greatly favored.

3. Model

The algorithm operates in an asynchronous system where each process has local variables, a set of guarded rules and a lossless FIFO link towards all the other processes. Processes do not know their neighbors a priori, so p can send a message to q if and only if the identifier of q is in a local variable of p. The capacity of each channel is bounded by an unknown constant, and like Afek and Bremler [1], we assume that this issue can be addressed by buffering a constant number of messages if necessary. The scheduler is nondeterministic and fair in the sense that any rule whose guard evaluates to true in an infinite suffix is eventually drawn.

Each process has a unique identifier in a domain $\mathcal{I}$ that is divided into two subdomains $\Pi \subseteq \mathcal{I}$ contains the identifiers of *correct* process, i.e. those that do not crash, and the other identifiers belong to crashed processes or to no process at all. However, a given process cannot know whether some $i \in \mathcal{I}$ is in Π or not. A total order $<$ on $\mathcal{I}$ is available, as on most large scale systems: for example, IP addresses are totally ordered.

To provide the processes with the ability to connect the whole system, each of them has a local device called an *oracle*. Each process can query its oracle as part of an action by calling the function get_peer(). The answer is an identifier in $\mathcal{I}$. The global condition on the set of all oracles in the system is that in an execution, if $\mathcal{S}$ is the set of all processes that query their oracles an infinite number of times, then all the processes in $\mathcal{S}$ eventually obtain all the identifiers of $\mathcal{S}$. This device is an adapted version of the concept of peer sampling service, tailored for the needs of the algorithm.

To make it possible to actually use this system, the processes need to be able to eventually decide whether a process is live or not. Since Fischer, Lynch and Patterson's theorem [5] states that the consensus problem is unsolvable in an asynchronous system where a crash is possible, we use failure detectors. Each process has one, following the definition given by Chandra and Toueg [2]. This local device provided a predicate suspect: $\Pi \mapsto \mathrm{boolean}$. In this papers, all the detectors are in class $\diamond\mathcal{P}$, which means that after a finite but unknown time, for all processes, $\forall p$, suspect(p) $\Longrightarrow$ process p is crashed.

The algorithm makes use of self-stabilization [3]. This technique, introduced by Dijkstra [3] and since then adapted to many different contexts [4], allows a system to recover from any sequence of transient failures. To account for this, the whole system, comprising the processes and the channels, is initialized arbitrarily. Then, the runs are failure-free.

4. The Algorithm

This is a simplified version of the algorithm. It is sufficient to understand how the tree is built. The complete algorithm is given in the previous paper [8].

The goal is to build a spanning tree whose degree is bounded by a constant δ. To make sure that there can be no

cycle, the algorithm enforces a global invariant: the identifier of each process is higher than those its children and lower than that of its parent.

Each process has local variables, an identifier ($myself$) and a set of guarded rules.

5. Model for the Probabilistic Study

The above model is well suited for a proof of convergence in all possible cases. However, for a probabilistic study of the convergence time, we switch to a simpler model.

First, we assume the existence of a mapping from Π to $\mathbb{N}^*$ that assigns consecutive integers to the processes. The process that has the lowest identifier in Π gets number 1, the immediately higher process gets 2, and so on. The highest process is numbered N. This mapping, which is normally not available in a large scale system, is only used for the presentation of the probabilistic analysis and not by the algorithm. We also assume that only the identifiers of live processes are drawn; this does not change fundamentally the results but makes the following explanations easier to understand.

Then, we switch to a new scheduler that uses a coarse-grained notion of rounds. Rounds are the time unit in the rest of this paper.

Definition 1 (Round)**.** A round is a minimal sequence of consecutive events during which each process executes its spontaneous guarded rule at least once and all the messages that this produces are received.

The existing proof of self-stabilization allows us to switch to this definition since it implies that the number of actions in a round is necessarily finite.

We define the start of round 1 as the first action in an execution and, recursively, the start of round $i > 1$ as the first action following the last action of round $i - 1$.

6. An Upper Bound on the Convergence Time

We divide the discussion according to the value of δ, the bound on the degree of the spanning tree we are building. Nevertheless, we follow the same guiding thread for the different cases. The idea, as proposed by Gouda [7], is to define a metric over the set of configurations. Indeed, usually, in order to prove that a randomized self-stabilizing algorithm $\mathcal{A}$ reaches a set $\mathcal{L}$ of legal configurations in finite time with probability 1 starting from an arbitrary initial configuration, one exhibits a potential function that measures the distance of any configuration to $\mathcal{L}$, such that this potential function decreases with non-zero probability at each step of $\mathcal{A}$. In each subsection, we define an appropriate metric (potential function) according to δ, the bound on the degree.

Variables
parent: $\Pi \cup \{\perp\}$ ($\perp$ means no value)
children: set of Π
Guarded rules
the spontaneous rule, in practice, is regularly triggered by the scheduler and mostly takes care of cleanup and connectivity.
true $\rightarrow$
    delete any incorrect child: greater than myself, cardinal of the child set bigger than δ, suspect child ;
    if father = $\perp$ (i.e. this process is root), pick a process given by the oracle and send it a message to try to merge the subtrees.
reception of a merge message
merge message available $\rightarrow$
    if this process is root and lower than the sender, it sends back a merge agreement message and take the sender as its new parent.
reception of a merge agreement message
merge agreement message available $\rightarrow$
    If possible, take the sender as a child. If there are already δ children, first delete a child lower than the sender. If all δ children are higher than the sender, pass the merge agreement down to one of the children.
link maintenance
true $\rightarrow$
    Send a keepalive message to parent and children
keepalive message available $\rightarrow$
    if the sender neither parent nor a child, take it as my new parent or child, if possible ; else send back a link rupture message. On reception of such a message, delete the sender from the children list or the parent field.

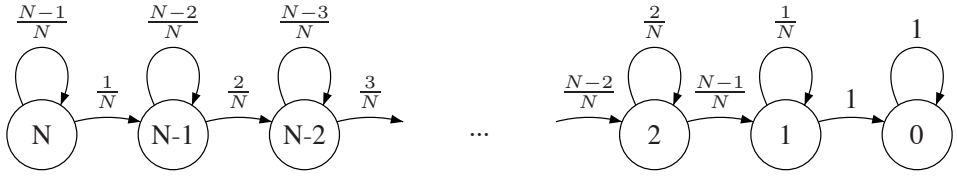

Figure 1. Model for $\delta = 1$

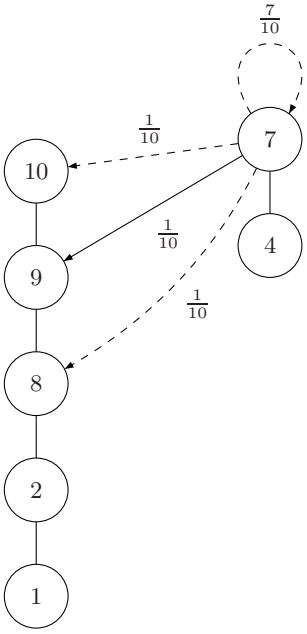

Figure 2. $\delta = 1, N = 10$, Number of processes at their right place$= 3$. Insertion of a subtree.

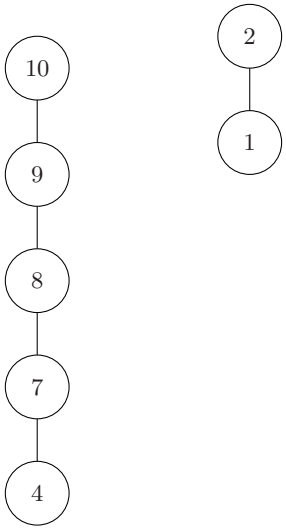

Figure 3. $\delta = 1, N = 10$, Number of processes at their right place$= 4$. After the insertion: the metric has decreased.

Once the metric reaches its minimum, the system has converged.

In two separate subsections, we consider first an oracle that uses a uniform distribution, then an oracle that follows a power law.

6.1. Uniform distribution

In this section, we assume that the oracle may choose each identifier with probability $1/N$.

6.1.1 Case $\delta = 1$

Since $\delta = 1$, each node may have only one son. Therefore, the main idea is that only one tree is a solution to the problem. It is in fact a chain of processes, ranked from the highest identifier to the lowest. We define that a process is at its *right place* iff 1) the root of its tree is Max and 2) its parent is the smallest higher process and its child is either $\perp$ or the highest lower process. Once a process is at his right place, it will never change position. We define the following metric: $\mu =$(Number of processes that are not at their right place).

Let us study the worst scenario: during each round, only

one process joins the final tree, so that the number of processes at their wrong place decreases by one at each round. We model the problem as shown in figure 1, where a *state* represents the number of processes that are not at their right place.

During each round, the process with the highest identifier that is not in the final tree queries the oracle and has a non-zero probability to join the final tree. For example, let us assume there are k processes at the wrong place. Therefore we know that there are already $N-k$ processes in the tree. When P_0, the process with the highest identifier among the $N-k-1$ processes not in the final tree, executes the algorithm, if the oracle answers one of the $N-k$ process with a higher identifier, then P_0 joins the final tree. Otherwise, nothing happens. Therefore, it is enough that the oracle chooses one of the $N-k$ process with a higher identifier for the metric to decrease. Such an event has probability $(N-k)/N$ to occur.

An example is shown on figure 2. We see that μ = 10-Number of processes at their right place = 10-3 = 7. Then process 7 sends a message to process 9. This means that after having queried its oracle, the answer was 9. According to the algorithm, the maximal degree of the tree being reached, process 9 looks at the identifier of his son. Its child has an identifier higher than 7. Therefore the new process does not replace its son; instead, the information is passed down to process 8. This one has a child with a lower identifier than 7, so a replacement is done. The result is shown on figure 3, where $\mu = 10-4 = 6$.

To compute the expected time for μ to reach its minimum, we introduce the following stochastic process: $\forall k \in \mathbb{N}, X_k$ = Number of processes not at their right place at time k.

Proposition 1. *$(X_n)_{n\in\mathbb{N}}$ is a Markov Chain.*

Proof sketch: It is easy to see that the number of processes correctly placed on the tree at time $t+1$ only depends on how many of them were at their right place at time t.

Remark 1. This ensures that the algorithm converges with probability 1.

Definition 2. The expected time for the chain to reach state k, knowing its initial position was l, is: $\forall k, l \in [0,\dots,N], T_k^l = \mathbb{E}[x|X_x = k \text{ knowing } X_0 = l]$.

Using these definitions, we obtain the following induction formula: $T_0^{k+1} = 1 + \frac{N-k}{N}T_0^k + \frac{k}{N}T_0^{k+1}$ which leads to $(T_0^{k+1} - T_0^k) = \frac{N}{N-k}$. After summing from 0 to $N-1$ (we notice that $T_0^0 = 0$), we have: $T_0^N = N\sum_{i=0}^{N-1}\frac{1}{N-i} \sim_{N\mapsto+\infty} N\log(N)$.

6.1.2 Case $\delta = N$

This is the second marginal case. We need to define a different metric: μ = (N − Identifier of the smallest root). The tree is built as soon as there is only one root, i.e. when the identifier of the smallest root is N, thus when $\mu = N - N = 0$. Again, we study the worst-case scenario, i.e. at each round only the smallest root joins the final tree. This is illustrated in figure 4, where a *state* represents the identifier of the smallest root. Assume k is the smallest root: when k queries the oracle, it is enough that it answers one of the $N-k$ higher identifiers for the metric to decrease. This justifies the probability transitions that appear on the figure.

We introduce the following stochastic process to compute the expected time for μ to reach 0: $\forall k \in \mathbb{N}, X_k =$ the identifier of the smallest root at time k.

Proposition 2. *$(X_n)_{n\in\mathbb{N}}$ is a Markov Chain.*

The proof is similar to that made in the case $\delta = 1$.

The computation of the convergence time is similar to what we did in the case $\delta = 1$. Using the previous definition: $\forall k \in [0,\dots,N]$ $T_N^k = \mathbb{E}[x|X_x = N \text{ knowing } X_0 = k]$, we obtain: $T_N^k = 1 + \frac{N-k}{N}T_N^{k+1} + \frac{k}{N}T_N^k$ which leads to $(T_N^k - T_N^{k+1}) = \frac{N}{N-k}$.

Therefore, the expected convergence time is $T_N^0 = N\sum_{i=0}^{N-1}\frac{1}{N-i} \sim_{N\mapsto+\infty} N\log(N)$.

On the example given in figure 5, we see that the process with the smallest identifier has a non-zero probability to join the tree.

6.1.3 Case $2 \leq \delta \leq N-1$

We now turn to the main case, that is $2 \leq \delta \leq N-1$. We define the following metric: μ = (Number of roots,identifier of the smallest root).

The minimum of this metric is (1,N). When it is reached, there is exactly one root, namely Max, and convergence is achieved. This metric cannot increase since it is not possible in the algorithm to drop two children simultaneously. We now show why it decreases. The idea is that during each round, either the first component decreases, or the second one does. If the number of roots decreases then, obviously, the metric decreases. Otherwise, a root tried to join a tree and took the place of another process. According to the algorithm, a process can only replace a child p with a process q s.t. $q > p$. This indicates that the new root necessarily has a smaller identifier than that of the former root. The second component of the metric decreases.

Figure 6 shows the possible cases for the insertion of a subtree into the main tree.

We can now compute the expected time before this metric reaches its minimum. The worst case goes as follows: the first component stays constant until the second one reaches its minimum, then the number of roots (the first component of the metric) decreases by 1 and the second reaches its "new" maximum, and so on.

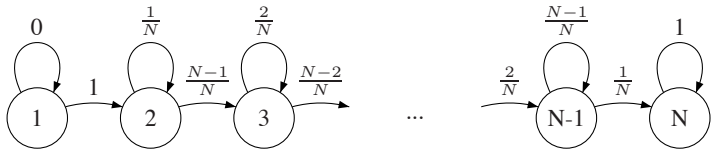

Figure 4. Model for $\delta = N$

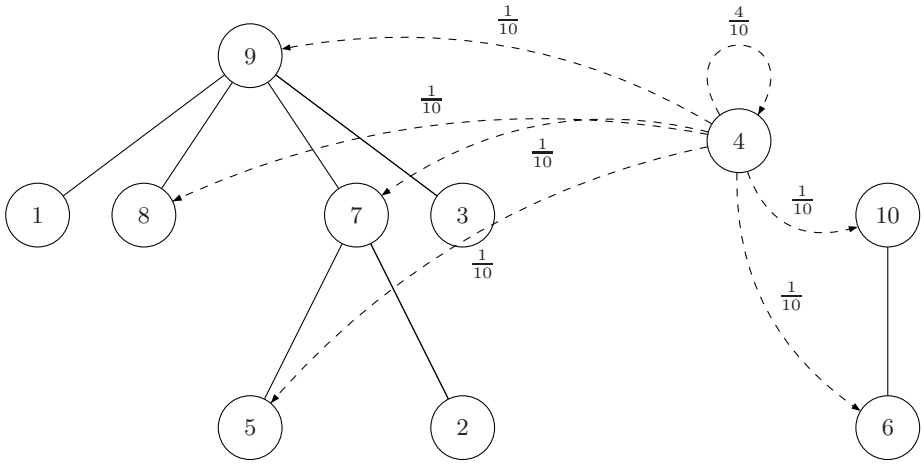

Figure 5. Example for $\delta = N$

Therefore, in order to compute the worst-case convergence time, we need evaluate the time needed for the second component of the metric to reach its minimum, i.e. the time after which no process smaller than the smallest root can become root. In order to study the convergence time, we split the calculations into two phases.

When we studied the case where $\delta = N$, we showed that any process k becomes the smallest root in the system in at most $(N-k)\log(N-k)$ rounds. Thus, since at most N processes will go through this number of rounds, the total convergence time is equivalent to: $N+\sum_{k=0}^{N}(N-k)\log(N-k) \sim_{N\mapsto\infty} N^2\log(N)$.

Proof sketch: The idea is to use the equivalence between sum and integral:

$\sum_{k=0}^{N}(N-k)\log(N-k) \sim_{N\mapsto\infty} \int_0^N x\log x \mathrm{d}x$.

We obtain: $\int_0^N x\log x = \left[x^2\log x\right]_0^N - \int_0^N xdx$, which is enough to lead to the result.

6.2. A more realistic distribution

In a state-of-the-art large scale system, the global knowledge of the whole set of process identifiers is distributed among all the processes. The usual device that connects the processes to the rest of the system is a *peer sampling service*. It provides each process with a small set of live process identifiers and ensures that the global knowledge graph is connected.

As a result, in the global overlay, each process has a small number of direct neighbors, which in turn know a small number of other processes. Therefore, a realistic oracle based on such a service should give each process p a very high probability of obtaining the identifiers of a small set of processes, which can be seen as *close* to p, and a very low, but non-zero, probability of obtaining any other process.

The choice of close processes follows practical requirements. It is desirable for large scale systems to be *self-organizing*, i.e. not relying on human intervention to build their overlay. They should also be *self-optimizing*, meaning that they improve their structure on their own to achieve

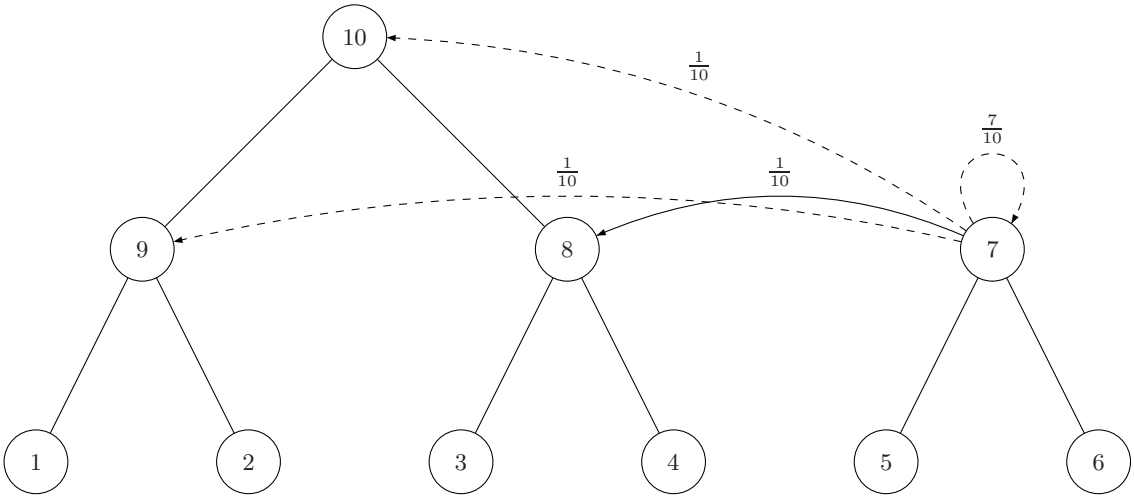

Figure 6. Insertion of the subtree rooted in 7 into the main tree

better performances. The notion of good performance is defined on a per-system basis: the metric to be minimized between any two virtual neighbors can be the number of hops in the underlying network, geographical distance, latency, or cost. An algorithm using the peer to peer system should thus, in turn, try to reduce its number of hops in the virtual topology.

In this section, the answers of the oracle are distributed as follows: $k \in [\![1, \lfloor \frac{N}{2} \rfloor [\![$ and $\alpha \in]0, 1[$ are given. If process i queries the oracle, it returns process j with probability p_{ij} such that $p_{ij} = \alpha/2k$ if $|i-j| \leq k \ mod(N), (1-\alpha)/(N-2k)$ otherwise.

The main idea is to give α a large value (close to 1) to introduce a bias in favor of the nearby processes.

In order to complete the complexity study, we must now compute the convergence time of the protocol assuming that the oracle follows this distribution. To simplify the calculations, we assume without loss of generality that k is 1.

6.2.1 Case $\delta = 1$

We use the same metric as for the uniform distribution: $\mu =$(Number of processes that are not in their right place). This measure converges toward 0. We now compute the convergence time.

We define $\beta_k = \frac{\alpha}{2} + \frac{N-k-1}{N-2}(1-\alpha)$.

Using the same technique as in the previous calculations, i.e. estimating the time to reach 0 knowing that we started from k we obtain that

$$T_0^{k+1} = 1 + \beta_k T_0^k + (1-\beta_k) T_0^{k+1}$$

So: $T_0^{k+1} - T_0^k = \frac{1}{\beta_k}$. By summation over k, we obtain

$$T_0^N \sim N ln \frac{2+\alpha}{2-\alpha}$$

We observe that convergence is achieved faster. Moreover, once a process asks the oracle for a new value, it has a greater chance to obtain exactly the right one (i.e. the immediately higher identifier), which decreases the number of delays in the algorithm.

6.2.2 Case $\delta = N$

Again, we reuse the same metric: $\mu =$(N-Identifier of the smallest root) in order to compute the convergence time.

The scenario is exactly the same as in the previous section. We compute by summation T_N^0. The convergence time towards N for μ is:

$$T_N^0 \sim N ln \frac{2+\alpha}{2-\alpha}$$

which is once again better than the convergence time in the uniform distribution case.

6.2.3 Case $2 \leq \delta \leq N-1$

This case is more complex, but we take again the same metric: $\mu =$(Number of roots, Identifier of the smallest root).

As usual, the second component of the metric first decreases to its minimum, then the first one decreases.

In the same way, the convergence time is equivalent to $ln \frac{2+\alpha}{2-\alpha} \sum_{k=0}^{k=N} (N-k)$, which is equivalent to

$$N^2 \ln \frac{2+\alpha}{2-\alpha}$$

	$\delta = 1$	$1 < \delta < n$	$\delta = n$
uniform distribution	$O(n \log n)$	$O(n^2 \log n)$	$O(n \log n)$
power law distribution	$O\left(n \log \frac{2+\alpha}{2-\alpha}\right)$	$O\left(n^2 \log \frac{2+\alpha}{2-\alpha}\right)$	$O\left(n \log \frac{2+\alpha}{2-\alpha}\right)$

Figure 7. Results summary

Here again, the convergence time is slightly better. This improvement in the convergence time is an encouraging result from a practical point of view since this kind of result distribution is easier to implement on actual peer to peer systems.

7. Conclusion

We studied the probabilistic convergence time of a self-stabilizing spanning tree algorithm for large scale systems. This contribution is important because the whole point of switching to a model that replaces the usual neighbor list with an oracle and a failure detector is scalability, but its main shortcoming was the difficulty of characterizing the behavior of an algorithm under a given oracle.

The results are summarized in figure 7. We first showed how the algorithm behaves under the standard hypothesis that a process could learn the identity of its peers following a uniform distribution. Then, we switched to the more realistic hypothesis that the underlying topology is a small world network in which peers are discovered following a power law, where convergence is achieved faster.

References

[1] Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA97)*, pages 111–120, 1997.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, March 1996.

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

[4] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

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

[6] Laurent Fribourg, Stéphane Messika, and Claudine Picaronny. Coupling and self-stabilization. *Distributed Computing*, 18(3):221–232, February 2006.

[7] Mohamed G. Gouda. The triumph and tribulation of system stabilization. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer LNCS:972*, pages 1–18, 1995.

[8] Thomas Herault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, and Joffroy Beauquier. A model for large scale self-stabilization. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2007.

[9] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[10] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[12] G. Winkler. P. brémaud: Markov chains: Gibbs fields, monte carlo simulation, and queues. springer verlag, 1999 - reviewed for metrika.

[13] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1):41–53, January 2004.