A Practical Study of Self-Stabilization for Prefix-Tree Based Overlay Networks

Vlad Acretoaie, Eddy Caron, Cédric Tedeschi

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Practical Study of Self-Stabilization for Prefix-Tree Based Overlay Networks

Vlad Acretoaie — Eddy Caron — Cédric Tedeschi

N° 7252

Avril 2010

Domaine 3

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7252--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

A Practical Study of Self-Stabilization for Prefix-Tree Based Overlay Networks

Vlad Acretoaie , Eddy Caron , Cédric Tedeschi

Domaine : Réseaux, systèmes et services, calcul distribué
Équipes-Projets Graal & Myriads

Rapport de recherche n° 7252 — Avril 2010 — 15 pages

Abstract: Service discovery is crucial in the development of fully decentralized computational grids. Among the significant amount of work produced by the convergence of peer-to-peer (P2P) systems and grids, a new kind of overlay networks, based on prefix trees (*a.k.a.*, *tries*), has emerged. In particular, the Distributed Lexicographic Placement Table (DLPT) approach is a decentralized and dynamic service discovery service. Fault-tolerance within the DLPT approach is achieved through best-effort policies relying on formal self-stabilization results. Self-stabilization means that the tree can become transiently inconsistent, but is guaranteed to autonomously converge to a correct topology after arbitrary crashes, in a finite time. However, during convergence, the tree may not be able to process queries correctly. In this paper, we present some simulation results having several objectives. First, we investigate the interest of self-stabilization for such architectures. Second, we explore, still based on simulation, a simple Time-To-Live policy to avoid useless processing during convergence time.

Key-words: Overlay Schemes, Fault-Tolerance, Time-To-Live, Self-Stabilization

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Une étude pratique de l'auto-stabilisation d'un réseau d'overlay structuré en arbre de préfixe

Résumé : La découverte de services est un aspect fondamental dans le développement de plates-formes de calcul décentralisées. La convergence des domaines des systèmes pair-à-pair (P2P) et des grilles de calcul ont été à l'origine de nombreuses solutions à ce problème. Parmi elles, les solutions à base d'arbres de préfixe, montrent des performances intéressantes. En particulier, l'approche DLPT (Distributed Lexicographic Placement Table), est une approche dynamique et totalement décentralisée. La tolérance aux pannes y est gérée par des algorithmes auto-stabilisants, qui assurent la convergence autonome du système vers une configuration fonctionnelle après un nombre et des types de pannes arbitraires, et ce en un temps fini. Toutefois, pendant le temps de convergence vers une configuration fonctionnelle, le système peut ne pas être capable de traiter correctement certaines requêtes (ici de découverte). Dans cet article, nous présentons des résultats de simulation ayant plusieurs objectifs. D'abord, nous essayons de quantifier l'intérêt de l'auto-stabilisation dans ce type d'architecture. Ensuite, nous explorons une politique simple basée sur un TTL (Time-To-Live) pour éviter des traitements inutils pendant cette période de convergence.

Mots-clés : Réseaux d'overlay, tolérance aux pannes, Time-To-Live, auto-stabilisation

1 Introduction

Grids connecting geographically distributed computing resources have become a low cost alternative to supercomputers. The convergence of communities of grid computing and peer-to-peer systems has produced numerous designs to make grid middleware for fully decentralized platforms. One crucial point in the design of such systems is the service discovery [18]. More specifically, the need for flexibility and complexity in the service discovery process led to the emergence of a new kind of overlays, based on tries, *a.k.a.*, prefix trees. These architectures usually support range queries, automatic completion of partial search strings and extend to multi-attribute queries.

The Distributed Lexicographic Placement Table (DLPT) approach [4, 6] is one of them, providing dynamic load balancing [5] and formal guarantees for fault tolerance [3], while most fault-tolerance for structured peer-to-peer networks rely on replication mechanisms, like [11]. Replication can be very costly in terms of computing and storage resources and does not ensure the recovery of the system after arbitrary transient failures (memory corruption, network disconnection, etc.). Within DLPT, an alternative best-effort approach, based on the self-stabilization paradigm [8], is used. A self-stabilizing system, regardless of the initial state of the processes and initial messages in network links, is guaranteed to converge to its intended behavior in finite time. The convergence of self-stabilization and P2P networks is recent [10]. Self-stabilization is more powerful than classical approaches of *stabilization* for instance used in DHTs like Chord [17] in the sense that it *formally* ensures the convergence to a correct configuration starting from an *arbitrary* state.

In our context, self-stabilization is a best effort approach starting after replication has failed, and the tree topology has become inconsistent (disconnected topology, cycles, wrong prefix relationship, etc.). The DLPT approach autonomously converge to a correct topology, after arbitrary crashes. However, during the convergence to a consistent topology, the tree may not be able, according to its current state, to process service discovery queries correctly, the algorithm to route queries being designed to run in a correct topology. In particular, queries may traverse cycles infinitely and never reach their destination. This problem can be easily addressed by introducing Time-To-Live (TTL) mechanisms to avoid useless overhead when processing queries in a faulty topology. The objective of the paper is twofold, based on results from intensive simulations: (i) show the relevance of using a self-stabilizing approach in such a context; and (ii) introduce and experiment simple TTL strategies to reduce the overhead during convergence.

The remainder of the paper is structured as follows. Section 2 presents the DLPT architecture and its related works. Section 3 deals with the self-stabilizing mechanisms provided by the DLPT and details the needs for a TTL mechanism during the convergence period. Section 4 provides the set of simulation results obtained and their interpretation for the targeted context. Finally, Section 5 concludes.

2 P2P Service Discovery

Early DHTs [14, 17] were the first step towards P2P resource discovery. Some researches went then into finding ways to improve the retrieval process over such networks, like introducing multi-attribute range queries [2, 15, 16]. A new kind of overlay, based on tries, has emerged. Trie-structured approaches outperform others in the sense that logarithmic (or constant if we assume an upper bound on the depth of the trie) latency is achieved by parallelizing the resolution of the query in several branches of the trie. For instance, Skip Graphs [1] are similar to a trie, and are built based on skip lists. More specifically based on prefix trees, Prefix Hash Tree [13] and P-Grid [7] were then introduced. PHT builds a prefix tree over the data set on top of a DHT. The tree is used as an upper logical layer allowing complex searches on top of any DHT-like network. P-Grid builds a trie on the whole key-space, each leaf corresponding to a subset of the key-space.

The *Distributed Lexicographic Placement Table* (DLPT) [4, 6] is based on a distributed prefix tree dynamically growing as services are declared by servers, as illustrated by Figure 1 giving an example of a tree growing with three services being sequentially declared: `DGEMM`, `DTRSM` and `DTRMM` from the BLAS library. The structure used is a particular prefix tree called *Proper Greatest Common Prefix* (PGCP) tree, defined as follows:

Definition 1 (PGCP Tree) *A Proper Greatest Common Prefix Tree is a labeled rooted tree such that the following properties are true for every node of the tree: (i) The node label is a proper prefix of any label in its subtree. (ii) The greatest common prefix of any pair of labels of children of a given node are the same and equal to the node label.*

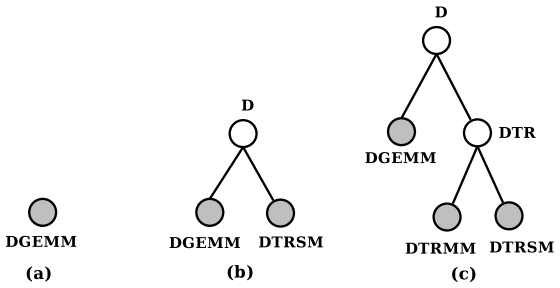

Figure 1: Construction of the prefix tree within DLPT.

A constant upper bound on both the degree of nodes and the depth of the tree can be assumed in such structures. Service registration is achieved by a server by issuing a registration query which can be sent to any node in the tree. The query is then routed inside the tree according to the name of the service to its destination, *i.e.,* the node labeled with this name. This node stores the information on the set of services registered under this name. If such a node does not exist, the query reaches the node with the label closest to this name, that triggers the creation of such a node. Tree nodes are dynamically, and in a decentralized way, distributed over the set of peers of the underlying

network, as detailed in [5]. Service discovery queries are routed similarly. As PHT and P-Grid, DLPT can efficiently process partial search string and more generally range queries. DLPT adopts load balancing mechanisms that take into account the heterogeneity of the capacity of peers and the dynamic evolution of the popularity of services requested. These load balancing mechanisms used in DLPT were inspired by load balancing techniques used in DHTs, and have shown interesting performances in DLPT context [5].

3 Fault-Tolerance in DLPT

As previously mentioned, one key feature of the DLPT compared to traditional P2P networks is self-stabilization. The topology is guaranteed to converge to a correct prefix tree, whatever the set of faults (fail stop or transient failures affecting communication, memory, etc.) raised in the system. We can discern two main types of errors addressed by the repair mechanism:

Memory corruption. Each node, to be able to decide the next routing step for a given query, maintains the information related to its neighbors (parent and children), serving as a routing table. This routing table is variable and then may be corrupted. For instance, a node p may consider q as one of its child while q considers $r \neq p$ as its parent. More simply, the corruption may affect copies of labels: one node may believe that its parent p is labeled by a given string s_p while in fact, the real label of p (as stored in p itself and incorruptible since constant) is $l_p \neq s_p$.

Topology corruption. The topology itself may be corrupted. For instance, after some arbitrary crashes, a node p whose label l_p is prefixed by the label l_q of one of its children q, even if routing tables are consistent with the broken topology (p considers q as a child, and q considers p as its parent, even if it should not be, according to Definition 1.)

3.1 Repair Algorithm

The algorithm uses the generic message passing paradigm. The convergence to a global property (to the topology of Definition 1) is achieved through a protocol constantly running on the nodes of the tree, detecting and repairing potential local inconsistencies in the tree resulting from failures. We now summarize this protocol. The protocol addresses both types of failures previously described. The first type of errors is addressed by simple ping-like messages containing their label exchanged between neighbors. The second type of errors, *i.e.,* affecting the topology itself, is based on the periodic execution of an algorithm on each node of the topology. This is a two-phases process, illustrated by Figure 2 from an initial faulty configuration (a) to the correct topology (f).

First phase: checking my parent. The first phase deals with parent maintenance and ensures that, eventually, the tree is a rooted connected tree. Let us consider a node p. We discern two cases: (1) If the label of p is the empty string, denoted ϵ, (see steps (d) and (e) on Figure 2), p tries to connect to another node q, also labeled ϵ. On Figure 2($d-e$), ϵ_1, root of the left tree, discovers ϵ_2, root of

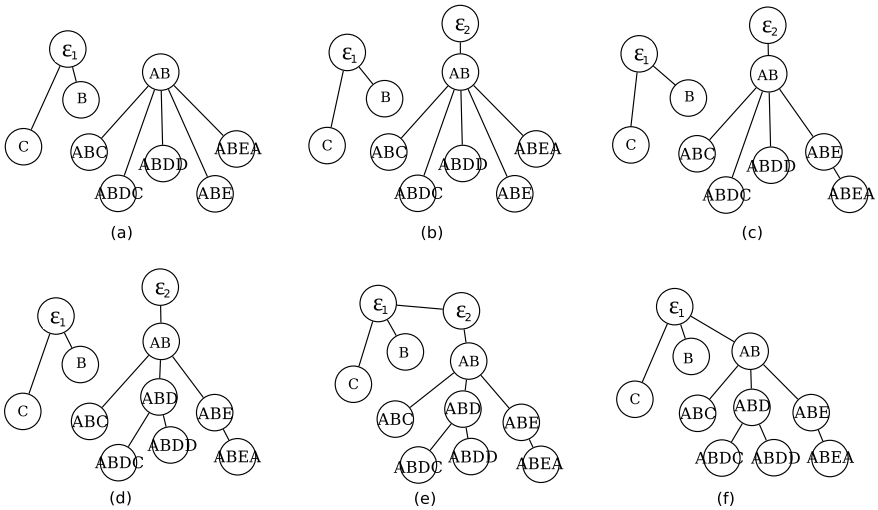

Figure 2: Self-Stabilizing DLPT protocol.

the right tree). q (ϵ_2 then becomes a child of p (ϵ_1). p and q then exchange some messages in order for q to update its parent's value. Since p and q are labeled identically, they will eventually merge, thus reducing the number of roots by one. (The merge process is explained below). (2) If p is not labeled by ϵ, a new node labeled ϵ is artificially created as the parent of p. On Figure 2($a-b$), the node AB creates the ϵ_2 node). The new node will, in a finite time, on its turn, execute the periodic rule, satisfying the previous case.

Second phase: checking my children. The second phase ensures that eventually, every set of children satisfies the definition of a PGCP tree (see Definition 1). This phase consists of three parts, each one addressing three possible inconsistencies affecting the set of children of one node p: (1) The set of children of p contains a node q whose label is the label of p. In this case, p and q must merge. This is solved by transferring information through a set of messages between p and q (on Figure 2($e-f$), ϵ_1 and ϵ_2 finally merge). (2) One child q_1 prefixes another child q_2 of p. In this case, the proper greatest common prefix of the labels of q_1 and q_2 is equal to the label of q_1. However, the greatest common prefix, according Definition 1, must be the label of p. By exchanging particular messages, q_2 then becomes the child of q_1. On Figure 2, step (c), node $ABEA$ becomes the child of node ABE. (3) There exists a pair (q_1, q_2) among the children of p such that the greatest common prefix g of their labels is greater than p's label. A new node must then be created, labeled by g, that will be the child of p and the common parent of q_1 and q_2. In Figure 2(d), node ABD is created. A detailed description and a comprehensive proof of this algorithm is provided in [3].

3.2 TTL Requirements

Introducing a TTL parameter for messages traveling through a network generally has one of two possible goals. The first is a performance requirement. For instance, in pioneering unstructured P2P networks such as KaZaA [12], searches are performed through crawling the network. Because, the diameter of the network is too high to be entirely scanned without making the performance of the platform collapse, a TTL limits the scope of the searches.

The second main purpose of TTL is to prevent messages from entering endless loops in case of a damaged network topology. This concept is widely used in logical as well as physical networks protocols. Many protocols, as common as the TCP/IP protocol, assigns Time-To-Live values to each data packet. In such networks, it is hard to choose an adequate TTL value: one that is big enough to allow messages to reach their destination and small enough to still be effective in killing stray messages.

In our case, we have to include a Time-To-Live in case of a broken prefix tree. The original key based routing algorithm used in DLPT for searches assumes the validity of the tree [4]. While a request is ensured to reach its destination in a finite time in a valid topology, we have no idea of how requests will traverse the tree when prefix relationships are not respected, even in the case where there is no cycles in the topology. It is not proved that the request is prevented from entering a cycle of forwarding steps. For instance, if routing tables are corrupted, a request may traverse the same subtree up and down, infinitely. If some loops appear in the topology, this will naturally lead to more cycles in the processing of requests, in particular when these loops are combined with routing table corruptions.

The particular nature of our network topology allows us to experimentally estimate adequate TTL values. The basic reason for this is the fact that the depth of a prefix tree has a higher bound imposed by the length of the keys it is built from. Studying the performance of queries under different faulty topologies and different types of keys aims to help choosing an adequate TTL value for our system. This study is showcased in the experiments of the next section.

4 Simulation Results

The experiments presented here were performed on the self-stabilizing PGCP tree structure introduced in Sections 2 and 3 and presented in greater depth in [3]. Our goals were to capture the performance of the self-stabilizing algorithm under different dynamic settings, test the reaction of the system to the introduction of a TTL parameter and empirically find an appropriate value for it.

4.1 Experimental Set-up

The prefix tree and self-stabilization algorithm are implemented in a message-passing discrete-time simulator[1] using the Python programming language. A simulation run can be broken down into three major phases.

[1] See http://www-sop.inria.fr/members/Cedric.Tedeschi/software.html

First, a graph structure with inconsistent links containing randomly picked node labels of a given type is generated. The following four types of keys were considered: (i) strings of random alphanumeric characters; (ii) common service names, *e.g.,* `S3L_mat_mult` from the S3L library of SUN); (iii) network addresses, for instance `fr.grid5000.orsay.node1`. Note that we use the reverse notation in order to allow automatic completion of partial addresses, *e.g.,* in a request issued from a user wishing to collect the addresses from the `fr.grid5000.orsay` cluster); and (iv) Unix-like file system paths, for instance `/usr/local/bin/emacs`. Second, The self-stabilization algorithm is applied on the generated graph, resulting in a valid prefix tree structure. Finally, a discrete time counter is started and incremented for a finite number of steps while at each time step random errors and queries are introduced in the prefix tree. Throughout the entire length of the simulation, the self-stabilization algorithm is constantly running and sending messages in an attempt to correct the errors introduced.

Note that the simulator only operates at the logical level of the prefix tree structure. The actual mapping of overlay tree nodes to physical peers in the system is not handled at this level, and our assumption is that one physical peer will host several logical tree nodes.

4.2 Self-Stabilization Performance

The time and number of messages required to achieve convergence to a correct topology are presented in Figure 3 and Figure 4. They both depend on the size of the tree, which is heavily influenced by the lexical structure of its keys. A prefix tree constructed from keys that exhibit a large number of common prefixes will have a greater depth than one constructed from very dissimilar keys. Moreover, the first tree is likely to contain a larger number of nodes, since a significant number of intermediary nodes will be introduced in order to preserve the two properties defining a PGCP tree (refer to Definition 1).

Both the time and number of messages required increase with the number of nodes. Notice however that the number of nodes featured on the x-axis of the two figures represents the size of the initial graph. This size does not coincide with the size of the PGCP tree that will be obtained, as this tree contains an arbitrary number of intermediate nodes introduced to preserve its properties.

Results indicate that random character keys lead to a better self-stabilization performance both in terms of time and messages, a consequence directly linked to the size and depth of the tree. However, encountering such keys in a real P2P system is unlikely. The most realistic case is that of keys representing common service names, in which the self-stabilization algorithm consistently performs better than in the cases of network addresses and file-system paths. The motivation is that the final two categories provide lexically structured labels by definition, while common service names only exhibit a coincidental structure (for instance, when a service provider labels its services starting with the same initials). The number of time steps required for stabilization seems to grow linearly but slowly with the number of nodes. Regarding the number of messages generated, the curves suggest that a really large number of messages are required. However, the absolute increase observed in the number of messages passed coincides with an increase in the number of nodes and links sharing these message loads, suggesting an acceptable communication overhead.

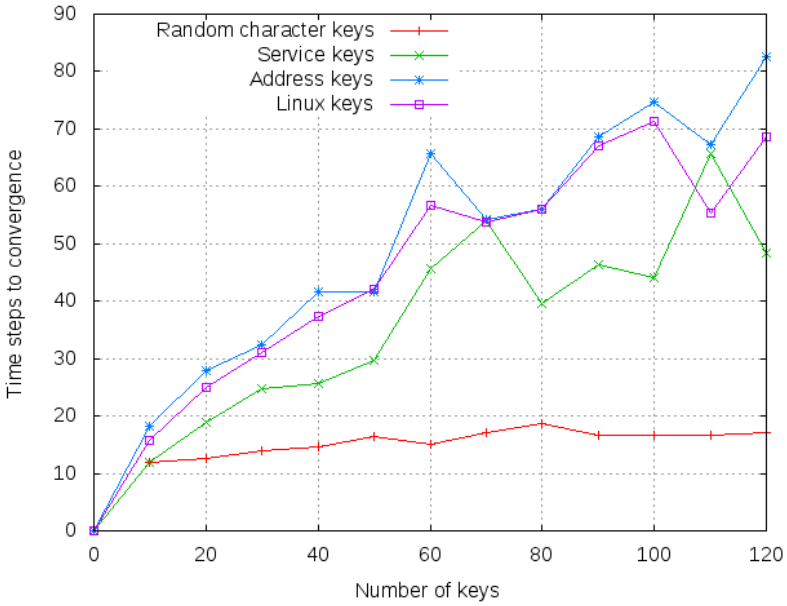

Figure 3: Convergence Time and Key Type.

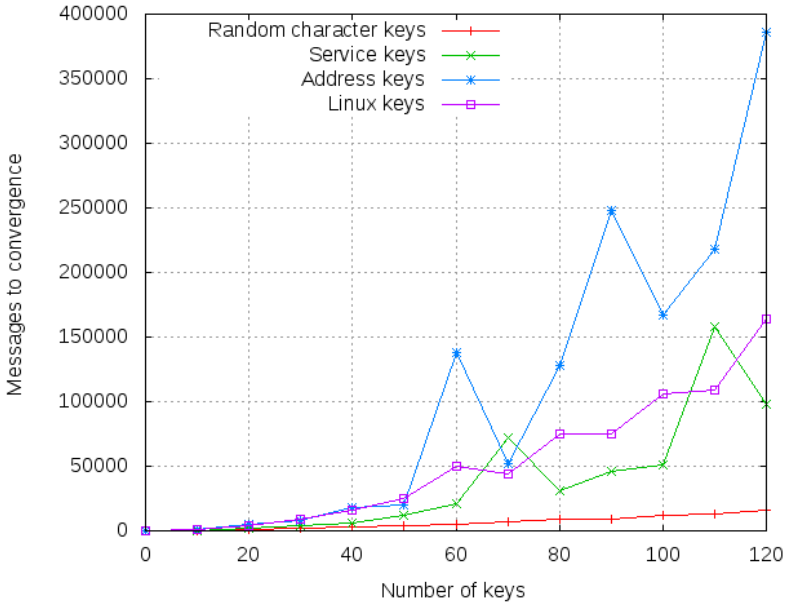

Figure 4: Amount of Messages and Key Type.

4.3 Query Satisfaction

The experiments in this category are meant to study the dynamic behaviour of the system. There are different parameters that influence this behaviour,

namely (1) the total number of errors introduced, (2) the distribution of errors over time, (3) the depth of the PGCP tree, and (4) the Time-To-Live of each query (*i.e.,* search for a key in the tree). The experiments were performed by launching a fixed number of queries at each time step of the simulation. A query that reached its destination in the tree was counted towards the successful query count of the step in which it was launched. The results were quantified by associating a Satisfaction Rate to each step of the simulation, defined as: $\frac{Q_{successful}}{Q_{sent}}$ where $Q_{successful}$ denotes the number of queries sent at the current time step that reach their destination and Q_{sent} denotes the total number of queries sent at the current time step. Three error distribution models were taken into consideration: (1) Periodical errors, (2) Completely random errors, and (3) Poisson-distributed errors. The PGCP tree was built from 150 labels representing service names, introducing a total of 120 errors according to different distributions throughout 60 time steps of execution.

Periodical Errors. In this scenario, errors were introduced in the PGCP tree periodically: 12 errors were introduced every 6^{th} time step. As shown in Figure 5 and Figure 6, the evolution of the satisfaction rate in time is relatively constant. However, for a TTL value of 5, the satisfaction rate remains quite small, while for a TTL value of 10 it approaches 1 (where almost all queries are satisfied). Naturally, for small TTL values queries do not manage to travel for a sufficient distance in the tree to reach their destination, even if that destination is available.

Randomly distributed errors. Although they are a good proof of concept, periodical errors are not close enough to what happens in a real system, where the exact times at which nodes become available or unavailable are non-deterministic. In view of this, the experiments were repeated with errors introduced at random time steps. The satisfaction rates related to TTL values are similar to the ones obtained with periodical errors, with slightly larger spikes on the plots due to a less even distribution of errors over time. Reasonable satisfaction rates are still only achieved for a Time-To-Live value of 10.

Poisson distributed errors. A practical and widely encountered distribution of stochastic events is the Poisson distribution, defined by the equation: $P(r) = \frac{e^{-\mu}\mu^r}{r!}$. It represents the probability of r events happening in unit time, with an event rate of μ. We consider the distribution to represent the probability of an inconsistency occurring in the PGCP tree during the time span of a simulation. The average value of the distribution, μ, is set to half the number of time steps in the simulation. This way, most inconsistencies are generated in the middle of a test run.

For a TTL value of 5, the satisfaction rate remains below 0.4 and the Poisson pattern does not reflect in the plots, proving that the satisfaction rate is reduced because of the TTL value, and not because of the errors. Instead, for a value of 10, the impact of errors on the satisfaction rate can be clearly observed. Satisfaction drops from a value close to 1 as soon as most the errors arrive (around the Poisson distribution mean value). Then, as the number of errors decreases, query satisfaction starts to increase again.

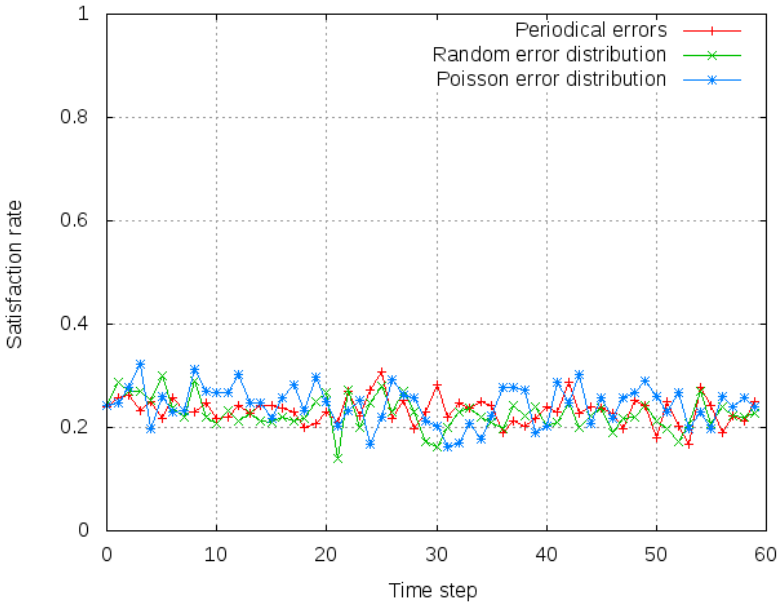

Figure 5: Satisfaction Rate for TTL=5.

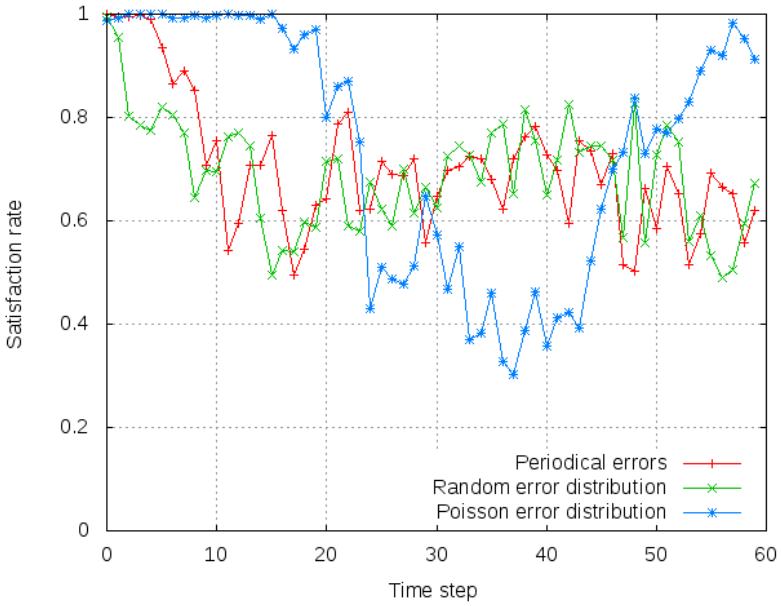

Figure 6: Satisfaction Rate for TTL=10.

4.4 Values for TTL

We have already established that small values for TTL lead to low satisfaction rates. This comes from the fact that, independently from the inconsistencies of

the topology, too few hops prevent from sufficiently exploring the tree. However, using very large values is not practical, as queries that have no solution in the tree will live longer than necessary. In fact, there is a threshold TTL value above which the satisfaction rate does not improve significantly and which is large enough to allow queries that can be solved to reach their destination. Our experiments have focused on finding this threshold for different types of labels, given the fact that the trie depth is constantly subject to change and thus less feasible as an upper bound for the TTL value. The PGCP tree was built from a number of 120 labels representing service names and errors were introduced according to the Poisson distribution. The only difference between the two experiments presented in this section is the nature of the labels. The results show an average satisfaction rate for Time-To-Live values ranging between 0 and 30, defined as $\overline{Satisfaction\ Rate} = \frac{\sum_1^{nb\_steps} SR_i}{nb\_steps}$ where SR_i represents the satisfaction rate at time step i.

Random character labels. Because such labels are completely random strings of characters generated from a given alphabet, the likelihood of any two keys having a common prefix longer than a few characters is quite small. This leads to a low depth of the tree, with many nodes having no children at all (*i.e.,* many leaf nodes) and with the root node having many unrelated children. The consequence is that the TTL values required for achieving a good average query satisfaction rate remain small, since there are few tree levels to traverse in search for a label. Interestingly, an adequate value for the TTL parameter in this case can be found around the value of 5. Remember that in the previous section this value was shown to be inadequate for keys representing service names. Also, the TTL value proves to be independent of the total number of errors introduced, as all the three plots in Figure 7 display similar shapes.

Common service name labels. The labels were picked randomly from a list of about 1000 known service denominations. Overall, the satisfaction rates are smaller than in the case of random character keys. This is justified by the greater height of the tree when service labels are used, which allows errors to occur at different levels, cutting off entire sub-trees instead of a single leaf, as opposed to the case of random character labels, where errors are most likely to affect leaf nodes (their number exceeds the number of non-leaf nodes). When considering service names as keys, an appropriate value for the TTL parameter is found around 9, as shown in Figure 8. Again, this threshold does not fluctuate significantly when the total number of errors introduced changes.

5 Conclusion and Future Work

We have presented an experimental performance study of a self-stabilizing prefix-tree based overlay network. We have focused on self-stabilization performance and query-delivery delay. Our results show that they are both heavily influenced by the nature of the keys used to build the tree, and that minimizing lexical structure optimizes convergence time. Then, we have evaluated how a classical TTL policy can improve the response time of a query in this context. The appropriate value to be assigned to this parameter is again influenced by

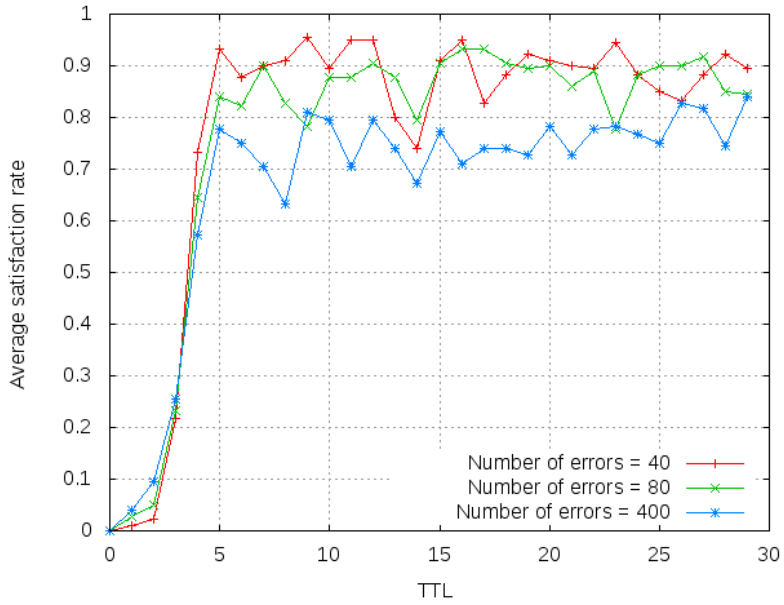

Figure 7: TTL and Random Labels.

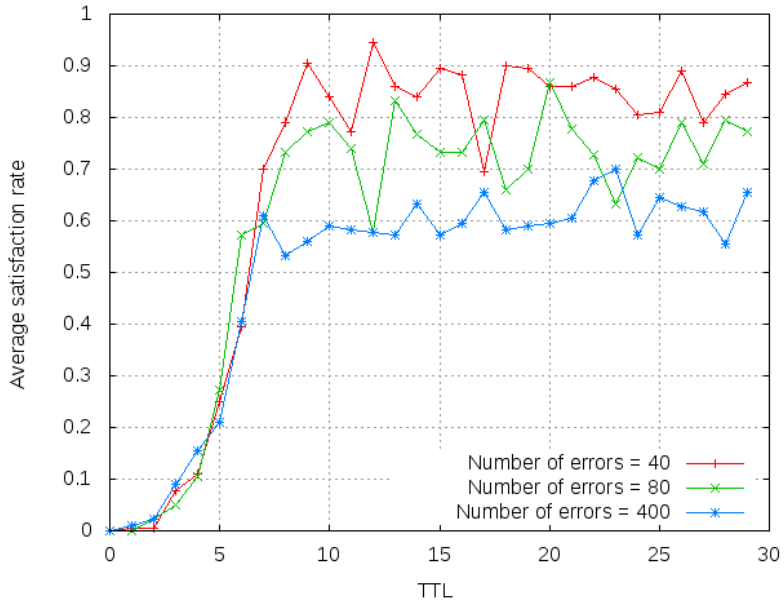

Figure 8: TTL and Structured Labels.

the nature of the keys. Globally, a TTL value between 5 and 10 hops seems to be enough for any key type.

Besides increasing the scale of the trees simulated, we have started the development of a real middleware prototype implementing DLPT concepts and its

deployment on the nation-wide Grid'5000 platform[2]. On the theoretical side, superstabilization [9], that combines self-stabilization and a *fast* recovery when only *few* failures occur, could be a promising research line to find new formal guarantees of availability in prefix-tree overlay networks.

References

[1] J. Aspnes and G. Shah. Skip graphs. *ACM Trans. Algorithms*, 3(4):37, 2007.

[2] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium*, 2004.

[3] E. Caron, A. K. Datta, F. Petit, and C. Tedeschi. Self-Stabilization in Tree-Structured Peer-to-Peer Service Discovery Systems. In *27th International Symposium on Reliable Distributed Systems (SRDS 2008)*, Napoli, Italy, October, 6-8 2008. IEEE.

[4] E. Caron, F. Desprez, and C. Tedeschi. A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. In *The Sixth IEEE International Conference on Peer-to-Peer Computing (P2P2006)*, pages 106–113, Cambridge, UK, September 6-8 2006.

[5] E. Caron, F. Desprez, and C. Tedeschi. Efficiency of Tree-Structured Peer-to-Peer Service Discovery Systems. In *5th International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P 2008)*, Miami, USA, April, 14-18 2008. IEEE.

[6] P. Chan and D. Abramson. A Scalable and Efficient Prefix-Based Lookup Mechanism for Large-Scale Grids. In *3rd IEEE International Conference on e-Science and Grid Computing, e-Science 2007*, Bangalore, India, December, 10-13 2007. IEEE.

[7] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *The Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[8] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[9] S. Dolev and T. Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[10] T. Hérault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A Model for Large Scale Self-Stabilization. In *IPDPS*, pages 1–10, 2007.

[11] S. Legtchenko, S. Monnet, P. Sens, and G. Muller. Churn-Resilient Replication Strategy for Peer-to-peer Distributed Hash-tables. In *The 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, volume 5873 of *LNCS*, pages 485–499. Springer, November 2009.

[2] http://www.grid5000.org

[12] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the Kazaa Network. In *The Third IEEE Workshop on Internet Applications (WIAPP '03)*, Washington, USA, 2003. IEEE Computer Society.

[13] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix Hash Tree: an Indexing Data Structure over Distributed Hash Tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, July 2004.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *Middleware 2001*.

[15] C. Schmidt and M. Parashar. Squid: Enabling Search in DHT-Based Systems. *Journal of Parallel and Distributed Computing*, 68(7):962–975, July 2008.

[16] Y. Shu, B. C. Ooi, K. Tan, and Aoying Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In *Peer-to-Peer Computing*, 2005.

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

[18] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-Peer Resource Discovery in Grids: Models and systems. *Future Generation Comp. Syst.*, 23(7):864–878, 2007.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399