Research Perspectives

Sébastien Tixeuil

March 21, 2006

Most of classical distributed fault tolerant techniques do not scale well. Using them would result to mechanisms that either consume too many ressources (memory, processing time), or are an overkill to solve the problem. To circumvent impossibility results in the area of self-stabilization, several paths were followed: restricting hypotheses on the faults that are likely to appear (either their nature or their geographic location), or restricting the kind of applications that can be solved. The frontier between problems that cannot be solved due to overwhelming costs, and those that admit reasonable solutions is still fuzzy. Several results show that there probably exists a tradeoff between the used resources and the capacity to tolerate failures, still a lot of further work is needed to obtain a clear vision of this tradeoff. In the next two sections, we further develop paths we believe are interesting research issues.

1 Theoretical side

Since its introduction in 1974 by Dijkstra [5], self-stabilization is based on a solid mathematical background. It is thus not surprising that various formalisms used in Mathematics, Automatics, and Computer Science were succesively used to prove self-stabilization: transfer functions [18], iteration systems [1], max-plus algebra **[10]**, rewriting systems [2], temporal [15] or high order logic [17], etc. The works we mentioned and conducted lead to other theoretical aspects related to self-stabilization.

1.1 Competitive self-stabilization

In "pure" self-stabilizing approaches, nodes cooperate to acomplish a common goal (*i.e.* satisfying a problem specification), in spite of an adversarial environment that tries to prevent stabilization. In Byzantine insensitive stabilization **[16]**, we distinguish two kinds of nodes, those that correctly execute the algorithm (the correct nodes), and those that try to fail stabilization (the Byzantine nodes). Here, all correct nodes collaborate, and all Byzantine nodes have unlimited resources to execute their actions.

It is likely that the model of self-stabilization with Byzantine failures is too extreme to match reality. For example, in interdomain routing in the Internet, nodes have a common goal (permit to exchange messages across the Internet) but also a local goal (like maximizing its own profit). It is then possible to refine the stabilization with Byzantine model in a new model that is not binary any more (correct *vs.* Byzantine), but rather united on a global goal, and competitive on the local goals.

1.2 Complexity and self-stabilization

There exist a large number of impossibility and lower bound results in distributed computing. For lower bound results, few results are specific to self-stabilization. The reason is twofold:

1. *memory lower bound* : if a "classical" distributed algorithm (*i.e.* non stabilizing) requires a certain amount of memory, then a self-stabilizing algorithm also requires this memo-

ry (it must behave properly when started from the initial configuration of the classical algorithm),

2. *time lower bound* : if a distributed algorithm requires a certain amount of time to solve a problem, then a self-stabilizing algorithm also (the stabilization time equals the *maximum* amount of time, over all possible system executions, including those that start from a correct initial state).

Thus, lower bound results of "classical" distributed computing translate directly to self-stabilizing algorithms, but the converse is not necessarily true. In particular, a number of problems that are impossible to solve in a self-stabilizing way (for example due to symetry in the intial configuration) can be easily solved in a properly initialized algorithm (restricting initial configurations so that such a symetry does not appear).

Recently, several distributed algorithms that approximate NP-complete problems have been designed. Self-stabilizing versions of some of them have jest appeared. There is usually a tradeoff between the *locality* of the algorithm (the amount of information about its neighborhood that it should known) and its efficiency (the quality of the proposed approximation). In a self-stabilizing context, the question that whether such a tradeoff exists remains open.

1.3 Systematic self-stabilization

In **[9, 10, 3]**, we have proved that a local condition on the local code executed by a distributed system could imply self-stabilization of the whole system under various hypothesis. In particular, depending on the considered model (high or low atomicity), different results were obtained (partial order in the case of high atomicity, total order in the case of low atomicity). A partial order permits to solve the *ordered ancestor list* problem, which in a strongly connected system (which corresponds to most actual systems) can be used (by the technique of [7]) to solve any static task. The fact that the same operator does not work properly in the low atomicity models does not imply, though, that there exist no such operator. An *ad hoc* algorithm was proposed in **[4]** for the ordered ancestor list in a low atomicity model. The question of whether there exists a universal operator (for static tasks) in the low atomicity models remain open.

Then, even if such an operator exists and shows that the approach is universal, maybe it is not the best operator to implement the solution to a particular static problem. Indeed, the memory space (in $O(n \log_2(n))$) and the amount of transfered information across the network is important compared to other specialized operators. Currently, the technique that permits to find an operator to solve a given problem, remains *ad hoc*. An interesting open question is whether there exists a way to systematically design an operator that implements a specified problem.

2 Practical side

A large number of the routing protocols that are currently used in the Internet are, to some extend, self-stabilizing. For example, the link state exchange protocol of OSPF (*Open Shortest Path First*, an intradomain routing protocol in the Internet) has been proved self-stabilizing by Nancy Lynch. Except in the routing domain, though, applications of self-stabilization are still limited. these limitations can be explained by various factors:

1. *hypotheses made about self-stabilizing systems do not apply to every actual system*: for example, self-stabilization assumes that processors never stop executing their code, but it is well known that a simple sequence of low level instructions (that could result from a memory corruption) can indefinitely hang some processors,

2. *self-stabilizing environments are not yet available for current software*: current operating systems upon which current software is executed were not designed nor proved self-stabilizing, so building self-stabilizing bricks on top of those foundations may seem artificial.

2.1 Self-stabilizing systems

To enable actual self-stabilizing distributed systems, two complementary approaches are possible:

the bottom-up approach : starting from the system foundations (hardware, operating system), we end up with applications that are based on self-stabilizing foundations. For example, the work of Shlomi Dolev follows this approach. In [6], they propose mechanisms to make a processor self-stabilizing, *i.e.* to guarantee that eventually, the processor infinitely executes the following sequence: *fetch*, *decode*, *execute*. Several approaches are described: it is possible to design from scratch a new processor, or to add to an existing processor an external harware mechanism (the *watchdog*) that verifies that the processors does not hang in some illegal state. Then, in [8], they have proposed the basis of a minimal self-stabilizing operating system (provided a self-stabilizing microprocessor) and considered several services (memory management, self-stabilization preserving compilation). While this work does give sound foundations for self-stabilizing systems, the disposal of a fully grown system that is usable for complex applications will still take at least several years.

the top-down approach : starting from the applications that are meaningful for current needs, we show that, under the assumption that underlying services are self-stabilizing, those new services are self-stabilizing as well. The verification of self-stabilizing properties in this context leads to many practical problems, since self-stabilization is usually compromized by few particular computations of the system, whose probability to occur in a real system is extremely low. Moreover, reproducing a particular execution that identified a problem in a self-stabilizing algorithm implementation decreases even more this probability.

We have designed and implemented a middleware platform, FAIL-FCI **[12]**, that should ultimately permit a top-down approach for constructing self-stabilizing systems. This middleware is developped in the context of several projects (the GrideXplorer project of ACI «Masse de Données», and the FRAGILE project of ACI «Sécurité et Informatique»), and has two main components:

1. *a fault scenario specification language* (FAIL, for *FAult Injection Language*) : this language permits to specify, using a formalism that mimics synchronized automata, scenarios that are meant to provide quantitative (every x seconds, a proportion y of the system has a z probability to experience a fault) and qualitative (once process p_1 at machine m_1 has executed line x of its code, then process p_2 at machine m_2 should be hit by a fault before executing line y) measures. In particular, this language permits cascading or epidemic faults where a causality relation exists between the first and subsequent faults.

2. *a distributed fault-injection middleware* (FCI, for *FAIL Cluster Implementation*) : this middleware operates between the operating system and the application under test. A nice property of our approach is that it is transparent to both the programmer and the user, because the source code of the application is not modified, and does not need to be recompiled.

The current status of FAIL-FCI enables to elaborate complex fault (or attack considering malicious faults) scenarios, and the fault-injection middleware permits to inject faults in two main kinds of applications:

1. native code applications (resulting from the compilation of *e.g.* C or FORTRAN source code),

2. some bytecode applications (currently, only Java bytecode is supported **[14]**).

For now, the set of faults that it is possible to inject with our tool are limited: crash faults (possibly followed by a restart from the initial state) and the suspend action (possibly followed by a resume action) that are meant to simulate fully asynchonous and faulty systems.

FAIL-FCI already permitted to reveal abnormal behavior in several applications (such as a global calculus middleware in **[13]**), and was noticed within the CoreGRID European Network of Excellence: for the second joint program of activities, a dedicated task "Fault injection and Robustness" was introduced. We plan to to further develop our tool. In particular, we plan to add new fault categories (memory corruption, resource premption), to test new kinds of applications (for example, Peer to Peer applications, that are well known to be scalable, have received little attention reguarding fault-tolerance and stress testing), and to permit fault-injection for different execution models (for example, MPI based applications). Another aspect worth investigating is the ability to replay real executions (modelled by execution traces) to evaluate application performance in a realistic context.

The ultimate goal of FAIL-FCI is to provide a tool that permits to run standardized benchmarks for fault tolerance and dependability in distributed systems.

2.2 Wireless Sensor Networks

Wireless sensor networks are one of the most obvious application areas for self-stabilizing actual systems. The two main reasons are as follows:

problems to be solved : many problems that are currently studied in the context of sensor networks can be modelled by graph problems, for which distributed or even self-stabilizing solutions exist. Moreover, the distributed nature of the solution is essential here since the envisioned size for sensor networks for the next years (dozens of thousands), it will be impossible to properly initialize such a system component by component according to the results of a sequential algorithm.

In the commonly used layered network model for wireless networks, distributed algorithms appear in the four upmost layers: data link, network, transport, and application layers. Most of the solutions that have been proposed, though, are for the data link and network layers.

For the data link layer, and most importantly the MAC (for *Medium Access Control*), several kinds of protocols can be used. Among them, the most common in wireless networks are *CSMA* (for *Carrier Sense Multiple Access*), *TDMA* **[11]** (for *Time Division Multiple Access*), or *FDMA* (for *Frequency Division Multiple Access*). In every case, the main goal is to permit access to the communication medium in spite of problems that compromise network performance (latency, throughput) or energy used to communicate (that is crucial in sensor networks). The main problem is due to collisions that occur when neighbor nodes concurrently try to access the radio medium to emit data: receiving nodes may then receive garbled or unusable data. In sensor networks, additional problems appear, like the fact that receiving a signal is almost as costly (in terms of energy) as waiting to receive a signal, this practical limitation induces algorithmic techniques that propose a tradeoff between latency and energy saving to communicate through the network.

By nature, TDMA and FDMA techniques are related to vertex or link coloring of some graph. As wireless networks we consider must be self-organised, this coloring cannot be predefined before the system is deployed, and must result from a distributed algorithm run by the system itself. Current distributed solutions to graph coloring problems show that theoretical bounds relate the quality of the coloring and the locality of the algorithm (that is strongly dependent of the used energy). Moreover, TDMA requires that local clocks are synchronized, which may require running supplementary distributed algorithms for synchronization.

Alternative algorithms based on solutions of graph problems can be used for the network layer. For example, it is possible to build an energy-wise efficient infrastructure by self-organizing the network in a hierarchical manner or by finding a subnetwork with interesting properties. The considered graph problems are then most related to dominating sets (sets of nodes that are capable to communicate with every other node in the graph), the goal being to make those sets as small and/or efficient as possible.

hardware specifics of sensor networks : wireless sensor networks are machines based on simple and low-cost components. Those machines support a limited set of devices, operating system services, and their operating system itself has a very small footprint. For example, TinyOS, the main operating system currently deployed in academic sensor wetworks, has a 3450 bytes code space , and a 226 bytes data space. This reduced size enables to study self-stabilization at the operating system level. Moreover, the fact that those networks are envisioned at a very large scale implies that fault tolerance must be part of their design from the begining. For most considered applications (data collection over a long period), non-masking solutions such as self-stabilization, are probably preferable, due to lower resource requirements, to masking approaches based on consensus and replication.

The currently sustained research effort in sensor networks will probably lead, within a few years, to large scale deployment of those (with several dozens of thousands nodes). In this context, it is not possible to manage those sensors or resume network behavior after failures manually. Techniques for large scale stabilization, until now developped in a theoretical manner, would be good candidates for such large scale deployment on real hardware systems. Indeed, they could lead to a unified and simple solution for both self-organization (made necessary by the large scale) and fault tolerance (that will frequently and continuously appear).

References

[1] Anish Arora, Paul C. Attie, Michael Evangelist, and Mohamed G. Gouda. Convergence of iteration systems. *Distributed Computing*, 7(1):43–53, 1993.

[2] Joffroy Beauquier, Béatrice Bérard, Laurent Fribourg, and Frédéric Magniette. Proving convergence of self-stabilizing systems using first-order rewriting and regular languages. *Distributed Computing*, 14(2):83–95, 2001.

[3] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. In *Proceedings of the Seventh Symposium on Self-stabilizing Systems (SSS'05)*, volume 3764 of *Lecture Notes in Computer Science*, pages 68–80, Barcelona, Spain, October 2005. Springer Verlag.

[4] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing*, 62(5):961–981, May 2002.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[6] Shlomi Dolev and Yinnon A. Haviv. Self-stabilizing microprocessor - analyzing and overcoming soft-errors (extended abstract). In Christian Müller-Schloer, Theo Ungerer, and Bernhard Bauer, editors, *Organic and Pervasive Computing - ARCS 2004, International Conference on Architecture of Computing Systems, Augsburg, Germany, March 23-26, 2004,*

Proceedings, volume 2981 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2004.

[7] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[8] Shlomi Dolev and Reuven Yagel. Toward self-stabilizing operating systems. In *15th International Workshop on Database and Expert Systems Applications (DEXA 2004), with CD-ROM, 30 August - 3 September 2004, Zaragoza, Spain*, pages 684–688. IEEE Computer Society, 2004.

[9] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, July 2001.

[10] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 293(1):219–236, 2003. Extended abstract in Sirrocco 2000.

[11] Ted Herman and Sébastien Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Proceedings of the First Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors'2004)*, number 3121 in Lecture Notes in Computer Science, pages 45–58, Turku, Finland, July 2004. Springer-Verlag.

[12] William Hoarau and Sébastien Tixeuil. A language-driven tool for fault injection in distributed applications. In *Proceedings of the IEEE/ACM Workshop GRID 2005*, page to appear, Seattle, USA, November 2005.

[13] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. Technical Report 1420, Laboratoire de Recherche en Informatique, Université Paris Sud, October 2005.

[14] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. In *International Workshop on Java for Parallel and Distributed Computing (joint with IPDPS 2006)*, page to appear, Greece, April 2006. IEEE.

[15] Sandeep S. Kulkarni, John M. Rushby, and Natarajan Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In Anish Arora, editor, *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 33–40. IEEE Computer Society, 1999.

[16] Toshimitsu Masuzawa and Sébastien Tixeuil. A self-stabilizing link coloring algorithm resilient to unbounded byzantine faults in arbitrary networks. In *Proceedings of OPODIS 2005*, Lecture Notes in Computer Science, page to appear, Pisa, Italy, December 2005. Springer-Verlag.

[17] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. In Ed Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 399–415. Springer, 1997.

[18] Oliver E. Theel and Felix C. Gärtner. An exercise in proving convergence through transfer functions. In Anish Arora, editor, *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 41–47. IEEE Computer Society, 1999.