Self-Stabilization

organized by

Prof. Dr. S. Dolev (Ben Gurion University, Israel)
Prof. Dr. A. Arora (Ohio State University, USA)
Prof. Dr. W.-P. de Roever (University of Kiel, Germany)

Preface

Distributed systems substantially improve our ability to compute and exchange information, as is evidenced by the dramatic success of the so-called World Wide Web. At the same time, distributed systems –and computer networks in particular– are hard to design, control, and maintain, as they consist of a variety of complex hardware and software components that are subject to faults and dynamic changes.

Self-stabilization has emerged as a promising paradigm for the design, control, and maintenance of fault-tolerant distributed systems. As its name suggests, self-stabilization enables systems to automatically recover from the occurrence of faults. Its essential idea is this: Regardless of what state a system is placed in, by virtue of being self-stabilizing, the system converges to desired behavior. Thus, even if faults cause the system to be placed in an arbitrary state, the system can eventually resume its desired behavior.

The field of self-stabilization is young and rapidly growing. To facilitate research in this field, experts in this field and in allied fields were invited to share their research interests and work with each other. The Dagstuhl Seminar on "Self-Stabilization" brought together thirty five researchers from seven different countries. The opening talk was given by Edsger W. Dijkstra, then an overview of the state-of-the-art and future directions was provided by Shmuel Katz. The talks that followed presented new results and directions:

- Formal methods for verification and specification of self-stabilizing algorithms,
- Use of the self-stabilization concept in the context of security and privacy,
- Integration with other fault models,
- Transient fault detectors,
- Design frameworks for achieving self-stabilization and other fault tolerances,
- Self-stabilizing algorithms and their time/space efficiency; impossibility results

The pleasant atmosphere of Dagstuhl was an important incentive for the lively interaction between the participants. The success of the seminar in stimulating new ideas and dialogue has led us to start planning the next seminar two years hence. We would like to thank all who contributed to the seminar, and in particular the encouragement we received from Professor Dr. Reinhard Wilhelm. The support of the TMR Program of the European Community is gratefully acknowledged.

The organizers

Shlomi Dolev
Anish Arora
Willem-Paul de Roever

Contents

The Influence of Self-Stabilization on Cryptographic Research

Moti Yung

We cover the development of cryptographic multi-party protocols under the notion of "mobile adversary" introduced in [PODC91, Ostrovsky-Yung]. The setting allows "perpetual faults" which move in the network while their presence is limited to a certain threshold t (e.g. minority of processors). Processors keep on failing and recovering

Under this adversarial setting we develop the notion of "proactive security" of protocols. This notion requires the processors in the system to take actions at well defined time units periodically to assure security and availability of the service provided by the processors. The actions are memory erasures and refreshment as well as memory recovery and processor-rejoins. The old local memory of processors are erased while the global value represented by them is maintained (this is a t-wise rerandomization of the memories).

We present generic "proactive secure distributed computation" (under mobile adversary), where the processors compute a known function (Boolean circuit) over private inputs and compute correct output while maintaining the secrecy of the inputs (e.g., secure ballot election). We also cover the more practical "proactive distributed public key systems" which is a current area of extensive research. In this setting, the private key (for signing or decryption) is distributed to the processors and a cryptographic operation requires a quorum of processors (for availability and control). The security is assured as long as within a well defined period, the adversary cannot break more than t processors; (note that in regular public key system, the breaking of one processor memory compromises the system). The setting allows us to add and omit devices as well as changing the threshold parameter t itself.

Introduction to Compositional Proof Methods for Program Correctness

Willem-Paul de Roever

Formal methods to specify and verify concurrent programs with synchronous message passing are discussed. I stress the development towards compositional methods, i.e., methods in which the specification of a compound program can be inferred from its constituents without reference to

the internal structure of those parts. Compositionality enables verification during the process of top-down design - the derivation of correct programs - instead of the more familiar a-posteriori verification based on already completed program code. I sketch the transition from non-compositional to compositional methods for concurrent programs, indicating the main principles behind compositionality, and discuss the main compositional frameworks using Hoare triples as basis.

Components for Fault-Tolerance: Theory and Application

Anish Arora
http://www.cis.ohio-state.edu/~anish

The thesis of our talk is that a fault-tolerant system is a fault-intolerant system encapsulated with tolerance components. We formally substantiate the generality of this thesis and present its application in the design of a scalable internet server that is stabilizing tolerant.

A Solution for a Peculiar Instance of the Consensus Problem

Augusto Ciuffoletti
http://www.di.unipi.it/~augusto/pub/dag_98331

We introduce a solution to a peculiar instance of the consensus problem. The motivation for this specific problem is in the fact that it models the part of a distributed clock synchronization problem which is not related with real time.

The solution is based on the diffusion of a request from the peripheral units to the privileged units, that respond with the value of a reference clock (to which they have a privileged access), which is diffused in the opposite direction. The diffusion is controlled by a hierarchical arrangement of the units, which operate following a 3-state self-stabilizing algorithm. The overall behavior is modeled as a series of waves, that propagate in the system, and that are periodically triggered by peripheral units.

One interesting property of the algorithm is the absence of statements explicitly aimed at recovering from non-legitimate states.

The correctness proofs outlined in the seminar have been carried out using the "Unity" tool (by Chandy and Misra).

New Tools and Lower Bounds for Self-Stabilizing Mutual Exclusion

Vincent Villain

The concept of *self-stabilization* was introduced by Dijkstra. He presented three stabilizing solutions to the mutual exclusion problem on the asynchronous rings with sense of direction. His solutions were later proven to be correct under both central and distributed daemon. One of the three algorithms also works on the linear arrays. The number of possible configurations in this algorithm is $2^2 * 4^{N \lceil 2}$, where n is the number of processors on the array. Another algorithm (among the three) presented in his paper works only on rings and has 3^N configurations, where n is the size of the ring. In his paper, a processor is considered to have a (mutual exclusion) privilege "if and only if it is enabled to make any move". Tchuente proved that with this definition of "privilege", the two algorithms of Dijkstra (as mentioned above) have the minimal number of configurations.

In this paper, we change the strict definition of privilege (as defined by Dijkstra) to the following: A processor has the mutual exclusion privilege "if and only if it is enabled to make a particular move". We first present a synchronous algorithm which needs only $n * 2^{N \lceil 1}$ configurations. Then we show that with a strongly fair central daemon, the lower bound is 2^N configurations. Finally, we prove that the lower bound of the number of configurations in the general case is $2^2 * 3^{N \lceil 2}$.

We present several mutual exclusion algorithms which match the above lower bound. The main idea behind all these optimal algorithms is the introduction of a new tool, called the "cleaner". In addition to the mutual exclusion algorithms presented in this paper, the usefulness of the cleaner has already been demonstrated by applying it to the design of efficient (both in terms of time and space) self-stabilizing algorithms for other applications, such as, the "depth first token circulation" and the "propagation of information with feedback"—both in tree networks. These algorithms can be used on general networks by combining them with any existing self-stabilizing spanning tree construction algorithms. We are currently working on the design

of these algorithms for the general networks without maintaining spanning trees.

Distributed Self-Stabilizing Algorithms

Tobias Vesper

We consider two versions of a load-balancing algorithm for a uniform ring of processors. The first version is designed for a shared-variable model. This version is self-stabilizing; meaning that if an error occurs that forces the system to an arbitrary state then the algorithm will eventually re-balance the system. Therefore, the system can tolerate any modification of variables.

In the second part of the talk we present a message-passing model. We derive a new load-balancing algorithm for this model. Though strictly speaking the algorithm is not self-stabilizing, it tolerates any modification of variables and of messages. We present a Petri net model and sketch the proof of the basic properties.

On the Relationship between Self-Stabilization and Fault Tolerance

Felix Gärtner

The paradigm of self-stabilization was introduced as a theoretical concept without a clear relation to fault tolerance. Traditionally, fault tolerance has taken the view that only a subset of nodes may be affected by permanent faults, while in self-stabilization *all* nodes may experience some form of transient faults. In this talk, I elaborated on the changing perception of self-stabilization as being a paradigm to also deal with permanent faults. Building on the different forms of fault tolerance (masking, non-masking, fail-safe) presented in the talk by Anish Arora, I identified self-stabilization as being an extreme form of non-masking fault tolerance that can also deal with a large class of permanent faults. A first characterization of the faults from this class was given: they must be eventually detectable, not destroy vital redundancy and must be "stable" for a sufficiently long period of time. A small example was presented, implications and problems were discussed.

Local Stabilization

Shlomi Dolev

Methods for detecting the occurrence of transient faults in distributed systems are presented. The techniques are *local* in the sense that faults are detected within a single time unit. The memory requirement for the implementation of these techniques will be discussed. First proving that there is no failure detector with a constant amount of memory for the distributed rooted tree construction task. Then presenting a failure detector for every distributed algorithm, using a large amount of memory. In the last part of the talk, we will show that different tasks requires different amount of memory for implementing their distributed failure detectors.
The talk summarizes joint works with Mohamed Gouda, Marco Schneider, Yehuda Afek, Joffroy Beauquier, Sylvie Deleat and Sebastien Tixeuil

Fault-Tolerance Patterns in Network Management Applications

Sandeep Kumar Shukla

A client server based *network management* application comprises of hardware and software components that cooperate to achieve the network management functionalities. Usually, these components are *agents* with specific data models, control logics, and user interfaces. These agents are organized in a tree-like hierarchy, with the *servers* at the top of the hierarchy, and the *clients* at the bottom. The clients are the agents that continually reflect the state of the network. Each agent has data models to represent the states of each network element of the network being managed.

In the *fault-management* arena of network management, we show that in most cases this hierarchy is actually a realization of the well known PAC (Presentation-Abstraction-Control) architectural pattern with the modification that all the bottom level agents are actually *observers* of the ongoing changes in the network state. So we describe this modified design pattern as **MOPAC** (multiple observer presentation-abstraction-control) architectural pattern. We then formulate an *invariant* that should hold in such an agent system for correct operation. Since, the network state changes *perpetually*, this invariant is formed with a time bound. For this time interval around any point in time, if the network changes are stopped, the data models for

a particular network element, at every agent should reflect the same state. This should be true of all network elements.

This formulation of the invariant leads to a formalization of the fault-tolerance requirements in such systems. Since the top level agents (servers) are prone to crash, and are critical components in the hierarchy, the fault class considered is the "crash of server agents". It is absolutely imperative that the invariant remains true even when such agents crash. Depending on whether we impose the strong requirement that the invariant should be true all the time, even during the time a crash occurs, or whether for a reasonable time period, we may tolerate the invariant to be falsified, we formulate two fault-tolerance design patterns, namely **MF-MOPAC** (*Masking Fault-Tolerant* MOPAC), and **NMF-MOPAC** (*NonMasking Fault-Tolerant* MOPAC).

These are the first design patterns in the literature that we know of, that has fault-tolerance requirement as the primal design force to be resolved. Some other design patterns such as *broker pattern, dispatcher pattern, persistence pattern* obtain some limited fault-tolerance as a byproduct of the respective strategies employed to resolve *other* design forces.

We also discuss some self-stabilization patterns which arise while considering the preservation of this invariant when the monitoring agents dynamically join or withdraw from the system, or when version updates take place at various agents.

Specification of Hybrid Components of Control Systems

Jüri Vain

Construction of formal sensor specifications is discussed within the context of stepwise refinement design discipline of control systems. The specification scheme developed constitutes a framework for systematic specification and verification of functional as well dependability properties and their dynamic during component's life cycle. The framework relies on abstract phase transition systems formalism. It defines main phases, phase invariants, and gives explicit characterization of gradual corruption process of sensors. As a case study an integrating temperature sensor specification derived according this scheme is described.

Deterministic and Self-Stabilizing Leader Election Protocol

Colette Johnen
http://www.lri.fr/~colette/

A protocol needs only *constant space* if required memory space on each processor depend on the processor degree and not on the system size. The interest of constant space protocols is that when the system changes its size (a global property, that cannot be locally checked), the local protocol implementations do not need to be changed. The implementation on a processor should be modified, only when the processor degree increases (a locally checkable property). We determine when it is possible to design deterministic self-stabilizing leader election algorithms requiring constant memory space.

First, we study uniform systems: processor does not have any identifier. We prove that the lower bound of memory space required by a deterministic self-stabilizing algorithm on unidirectional, uniform prime-size rings is at least $log(N)$ (N being the ring size).

Then, we study id-based systems: processors have distinct hardware identities that cannot be corrupted. We prove that if there is a self-stabilizing algorithm on id-based systems where the processor memory space is constant and the id-values are not bounded, then there is a self-stabilizing algorithm on uniform systems requiring constant memory size. Thus, the decidability results obtained on uniform rings can be extended to that kind of id-based rings.

Finally, we study the problem in the case of id-based rings where the processor memory space is constant and the id-values are bounded: we give a silent algorithm. A self-stabilizing algorithm is *silent* if once the system is stabilized, the values of local variables are fixed (write operations are totally eliminated); thus, the processors will only check their neighbor states. The silence property of self-stabilizing algorithms is a desirable property in terms of simplicity and communication/CPU overhead.

Randomized Self-Stabilizing Leader Election

Joffroy Beauquier

We present a new self-stabilizing leader election algorithm in rings of any size, that used O(log n) bit per process if n is a factorial, and O(log (log n))

bit on average. The proof uses a new modelization of the daemon, through the notion of restricted execution forest and presents a technique that can be used for a large class of randomized algorithms. The aim of the model is to separate randomization (in the protocol) and non-determinism (in the daemon). We prove that the algorithm is space optimal.

This work is common with Maria Gradinariu and Colette Johnen.

Evaluating Self-Stabilization

Shmuel Katz

The goals, problems, achievements, and remaining challenges of self-stabilization were analyzed in this talk–obviously a personal perspective. The overriding goal in this area is seen as gaining acceptance of self-stabilization as a standard form of fault-tolerance. To achieve this, developing algorithms, transformations, complexity measures, and clearly stated models of computation are crucial. To these should be added the goals of compositionality and modularity of both algorithms and correctness reasoning.

Among the problems in this area are the common perception of self-stabilization as too strong a fault-model for distributed systems. Both data and control are arbitrary in an initial state, with nothing safe from corruption, a stable state cannot be reliably identified from within the system, and the algorithms are inherently non-terminating. Other problems with self-stabilization are self-inflicted by the research community: papers that ignore significant differences in computational models, use unstated limiting assumptions, and advocate problematic complexity measures. Beyond these obstacles to acceptance, self-stabilization is seen as 'different' from other types of faults, orthogonal to crash, send-receive, or Byzantine faults.

After considering alternative definitions of the term, and the variety and sensitivity of related models, the value of general techniques was emphasized. Among these are *prodding*– having some process able to send a message without receiving one first, *stamping*– adding process ids of those processes that receive information before passing it on, and *flushing*– removing error-filled initial messages by guaranteeing that fresh round numbers are generated. Especially valuable are techniques for *composing* self-stabilizing algorithms. One way to verify a parallel composition is to show that each component does not interfere with the proof of correctness of the other component.

Such techniques should be reused and further developed, precise proof schemas and robust complexity measures should be used, and appropriate application areas should be identified to allow fully integrating self-stabilization into general fault-tolerance.

Self-Stabilizing Depth-First Token Circulation in Rooted Networks

Ajoy Kumar Datta

http://www.laria.u-picardie.fr/~petit/publi/DJPV98.ps.gz

We present a deterministic distributed depth-first token passing protocol on a rooted network. This protocol uses neither the processor identifiers nor the size of the network, but assumes the existence of a distinguished processor, called the root of the network. The protocol is self-stabilizing, meaning that starting from an arbitrary state (in response to an arbitrary perturbation modifying the memory state), it is guaranteed to reach a state with no more than one token in the network. Our protocol implements a *1-fair* token circulation scheme, i.e., in every round, every processor obtains the token once. The proposed protocol has extremely small state requirement—only $3(\Delta + 1)$ states per processor, i.e., $O(log\Delta)$ bits per processor, where Δ is the degree of the network.

The protocol can be used to implement a fair distributed mutual exclusion in any rooted network. This protocol can also be used to construct a DFS spanning tree.

Self-Stabilizing Timestamps

Uri Abraham

Messages are often timestamped. In a fax, the timestamp includes the date and exact time of the day, and in a book only the publication year, but in all cases this information guides the reader in choosing and processing the data. The antiquarian may choose the oldest book, and the student the newest edition, but in the general timestamp protocol the reader (called scanner) returns *all* the messages in their issuing order.

A new kind of timestamps is introduced in this paper: it is weaker than the regular one and is hence easier to construct. We build these weaker timestamps and our construction yields a self-stabilizing protocol (with bounded

values). It turns out that these weak timestamps suffice for implementing mutual-exclusion, ℓ-exclusion, and atomic register protocols. Hence we have self-stabilizing solutions for these protocols. I believe that these weaker timestamps may be efficiently used to replace (unbounded) regular timestamps in some other protocols. Indeed, the weak timestamps can also be used to implement regular timestmps, and hence we get here self-stabilizing (regular) timestamps.

A Foundation for Secure Computing

Mohamed Gouda

We argue that three fundamental concepts of stabilization theory are adequate to explain system security. These three concepts are closure, convergence, and protection. Our argument is based on the view that a secure system has good states and bad states such that the following three conditions hold. First, the set of good states is closed under system execution, and the set of good and bad states is closed under both system execution and adversary interference. Second, any system execution that starts at a bad state eventually converges to a good state. Third, each critical variable of the system is protected from being updated, as a result of some adversary interference, during any system transition from a bad state or during any system transition into a bad state. We use these concepts to state and verify the security of several systems, including one for transmitting data between two processes.

Synthesis of Self-Stabilizing Programs with Many Similar Processes

Paul Attie

Methods for synthesizing concurrent programs from temporal logic specifications eliminate the need to manually compose a program and manually construct a correctness proof. These methods rely on some form of explicit state enumeration and are therefore subject to state explosion. In previous work (joint with Allen Emerson, University of Texas, Austin) we avoided state explosion by constructing explicit products of small numbers of processes only. Our method then generates an arbitrarily large (i.e., parameterized) program syntactically. In this talk, we outline an extension of the method so that the synthesized program is self stabilizing.

Formal Verification of Stabilizing Systems

Michael Siegel

The talk presents a link between too formerly rather disjoint research areas: formal verification and stabilization. After a brief introduction to the principles and the design of stabilizing systems, a fully syntactical formal framework for phased, i.e. modular, verification of stabilizing systems is presented.

This framework is based on fair transition systems, linear temporal logic and refinement theory. It replaces the hitherto informal reasoning in the field of stabilization and constitutes the basis for machine-supported verification of an important class of distributed algorithms.