Self-stabilizing Reconfiguration

Shlomi Dolev  
Ben-Gurion University  
dolev@cs.bgu.ac.il

Ioannis Marcoullis  
University of Cyprus  
imarco01@cs.ucy.ac.cy

Chryssis Georgiou  
University of Cyprus  
chryssis@cs.ucy.ac.cy

Elad M. Schiller  
Chalmers Univ. of Technology  
elad@chalmers.se

CCS Concepts
• Software and its engineering → Middleware;

Keywords
Self-stabilization; reconfiguration; fault-tolerance

1. PROBLEM DESCRIPTION

We consider distributed systems that work in dynamic asynchronous environments, such as a shared storage system [8]. A configuration, is a set of active processors (servers or replicas), typically used to provide services to the system users. Over time, a given configuration can gradually lose members due to voluntary leaves and stop failures, rendering service provision difficult and eventually impossible. As a consequence, there is need to replace the current configuration with a new set including newly arrived processors and with departed processors removed, i.e., perform a reconfiguration. Over the last years, protocols for such reconfiguration middleware following different reconfiguration techniques have produced a wealth of research results, which are mainly related to state machine replication and shared storage (e.g., [1, 2, 6, 7]). In this work, we look into the implicit assumptions of the above research directions and propose new protocols that have significantly stronger fault tolerance properties.

Existing reconfiguration solutions are based on starting the system in a consistent configuration, in which all processors (both configuration members and other users) are in a predefined initial state. Starting from that state, the system must strive to preserve consistency in order to exhibit the desired behavior. Moreover, it is usually assumed that unbounded storage is available and that the churn rate of joins and leaves does not exceed some predefined bound. Nevertheless, large-scale asynchronous message passing networks are subject to transient faults (arbitrary changes in local variables or program counter) due to hardware or software temporal malfunctions, short-lived violations of the assumed churn rates or violation of correctness invariants, such as the uniform agreement among all active processors about the current configuration. These violations may, by the arbitrary corruption of the system’s state, and possibly render the system unavailable, requiring human intervention to recover. Fault tolerant systems that are self-stabilizing [3] can automatically—without external interference—recover after the occurrence of transient faults, as long as the program’s code is still intact. Replicated state machines providing cloud services such as Chubby, ZooKeeper, Raft and Doozier are unfortunately not self-stabilizing, and can be driven to a state where no functionality will be obtained whatsoever. Our self-stabilizing design is the futurist solution for such automatic recovering services.

Contributions. In this work, we present a middleware service that achieves reconfiguration and is the first, to our knowledge, that is also proven self-stabilizing. Namely, the service automatically recovers the system from transient faults, such as temporary violation of the predefined churn rate or the unexpected activities of processors and communication channels. Contrary to most of the existing non-stabilizing solutions, we only assume a bounded local storage and message size. Additionally, the service can install a configuration even in the case where the majority or even all current configuration members have collapsed. This is a case that many of the existing solutions do not address. Our self-stabilizing solution regains safety automatically by assuming temporal access to reliable failure detectors. Due to space limitations, we only provide an overview of our work. Full details (including the detailed description of the system, its algorithms and their analysis) can be found in [5].

2. THE SERVICE

The reconfiguration service is modular, composed of two protocols: the (i) Reconfiguration Stability Assurance (recSA) module and (ii) the Reconfiguration Management (recMA) module. The service is also complemented by a joining protocol. Figure 2 depicts the interaction between the modules and their interaction with the application.

The recSA layer of a processor exchanges messages with the other processors to guarantee that locally known configurations do not form a conflict (i.e., two processors having different configuration sets), since such a conflict is the result of a transient fault. In the case of conflict, the recSA protocol exchanges messages with other processors to make the conflict known and establish a new configuration. The module also provides information to the other modules and to the application about the current configuration and on whether a reconfiguration is taking place.
The reconfiguration service can be used to provide a robust set of processors to applications that may serve a multitude of tasks. In our work, we demonstrate the usability and modularity of our self-stabilizing reconfiguration scheme by demonstrating how to develop various self-stabilizing dynamic participation protocols from the corresponding self-stabilizing protocols designed for static systems. In particular, we propose a labeling service for providing a bounded self-stabilizing labeling scheme leading to a multi-purpose counter increment service; a self-stabilizing virtual synchrony service that can implement self-stabilizing state machine replication, or a self-stabilizing distributed shared storage service. These services are derived by combining our reconfiguration scheme and joining mechanism with the corresponding self-stabilizing services developed for static membership systems in [4].

4. IMPLEMENTATION & EVALUATION

We are currently moving towards the implementation of our reconfiguration service and of the complementary joining protocol. This will lead to an evaluation by comparison with non-stabilizing reconfiguration implementations, in order to estimate the practical costs that self-stabilization may incur in terms of the time it takes to converge to a single configuration, and also whether scalability is affected. The service can be deployed on a suitable network testbed, but it is still challenging to effectively emulate transient faults. It is, nevertheless, essential to benchmark the time that the protocol takes to stabilize, i.e., to return the system to its desired behavior after the transient fault has seized. This will provide a good indication of the tradeoffs between having a valuable fault-tolerant feature like self-stabilization, against the increased cost in message exchange and some (possible) delays in reconfiguration time. We should note that the cost of stabilization is tunable. One may choose to reduce stabilization related communication at the expense of slower stabilization.

A simulation, on the other hand, that only focuses on communication costs cannot capture the benefits of having a service that can automatically recover from state corruption. As long as the infrastructure and program are unaffected, the proposed service eventually returns the system to its desired behavior, contrary to other such solutions that can provide no guarantees in such scenarios.

Acknowledgments

The work of Ioannis Marcoullis is partially supported by a Doctoral Scholarship program of the University of Cyprus.

References

[1] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. J. ACM, 58(2):7, 2011.

[2] K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. MSR-TR-2010-151, Microsoft Research, 2010.

[3] S. Dolev. Self-stabilization. The MIT press, 2000.

[4] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing virtual synchrony. In Proc. of SSS 2015, pages 248–264.

[5] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller. Self-stabilizing reconfiguration. CoRR, abs/1606.00195, 2016.

[6] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. Dist. Computing, 23(4):225–272, 2010.

[7] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. SIGACT News, 41(1):63–73, 2010.

[8] P. M. Musial, N. C. Nicolaou, and A. A. Shvartsman. Implementing distributed shared memory for dynamic networks. Commun. ACM, 57(6):88–98, 2014.