Reconfiguring Self-stabilizing Publish/Subscribe Systems

Michael A. Jaeger*, Gero Mühl**, Matthias Werner, and Helge Parzyjegla***

Communication and Operating Systems Group
Berlin University of Technology
Einsteinufer 17, 10587 Berlin, Germany
{michael.jaeger, g_muehl, m_werner, parzyjegla}@acm.org

Abstract. Recent work on self-stabilizing routing in publish/subscribe systems showed that it is feasible to automate reconfigurations in case of faults by enabling the system to recover from arbitrary transient faults. In this paper, we discuss how to incorporate planned reconfigurations of the broker topology into self-stabilizing publish/subscribe systems without service interruption. We present an algorithm that uses a coloring mechanism to enable the system to be automatically switched from one system configuration to another. The colors thereby synchronize the broker overlay and the publish/subscribe routing layer.

1 Introduction

A publish/subscribe (pub/sub) system consists of brokers and clients. *Brokers* connect to other brokers to form an overlay network and to provide the *event notification service*. *Clients* connect to one broker and *publish notifications* or *subscribe to filters*. The broker overlay network routes published notifications to all brokers with clients that are currently subscribed to a matching filter.

Recent work on fault tolerance in the field of pub/sub middleware has shown that self-stabilization is feasible on the pub/sub routing layer [6] (for the sake of readability, we will use "routing" in the following when we actually mean "pub/sub routing"). Self-stabilization is an elegant mechanism for gaining fault tolerance. However, the solutions presented for pub/sub routing do not yet explicitly deal with reconfiguration [8,10], although managing these systems has to include reconfiguration of the broker overlay topology. In many self-stabilizing systems, reconfigurations are treated as faults and the system tries to "recover" from them. In contrast to this, our approach is to hold a "shadow" broker overlay topology that has already implemented the reconfiguration, to subsequently build up "shadow" routing tables on the pub/sub routing layer, and to finally switch the system atomically from one correct configuration to the next one using a coloring scheme. Thereby, we avoid notification and (un)subscription loss as well as duplication during reconfiguration.

* Funded by Deutsche Telekom Stiftung.
** Funded by Deutsche Telekom.
*** Funded by Deutsche Forschungsgemeinschaft (DFG SPP 1183 Organic Computing).

R. State et al. (Eds.): DSOM 2006, LNCS 4269, pp. 233–238, 2006.

2 Related Work

There are only a few publications in the area of self-stabilization that deal with reconfiguration issues explicitly. Most authors treat them as faults that will eventually stabilize like Dijkstra did in the initial paper on self-stabilization [1]. The concept of *superstabilization*, introduced by Dolev and Herman, has a more differentiated view by explicitly considering a certain class of topological changes [3]. Superstabilizing protocols are self-stabilizing and additionally require that if one change of this class occurs a *passage predicate* holds until the system is stable again. The passage predicate is usually weaker than the correctness predicate but is supposed to be strong enough to be still useful. In contrast to our approach, topological reconfigurations are supposed to happen immediately without any announcement and can thus not be delayed as we assume here. The concept of *fault containment* as described by Nelson [9] and applied to self-stabilizing protocols by Ghosh et al. [5] also tries to maintain service availability by keeping the effects of faults (or reconfigurations) locally bounded. Although fault-containment can dramatically reduce the effects of reconfiguration on the system in whole there is still an interruption of the service—at least in those parts of the system that are directly affected by the reconfiguration.

3 Assumptions and Model

As starting basis, we build on a model for self-stabilizing pub/sub systems developed in previous work [8]. Basically, we assume a hierarchical routing algorithm based on an acyclic broker topology with bidirectional FIFO links connecting individual nodes. Furthermore, there is an upper bound η on the number of brokers as well as a dedicated *root broker* R, which is globally known within the system. Considering self-stabilization, we require that the brokers' routing tables can be rebuilt from an *initial routing configuration*, which is empty for many routing algorithms [7], and that the routing algorithm bases its routing decisions solely on the contents of the routing table and the notification to forward.

Such a pub/sub system works *correctly*, if it meets the following two requirements [8]: (i) every client receives only the published notifications it has subscribed for (without duplicates) and (ii) every subscription becomes active after finite time, from which on the client receives every published notification matching its subscription until it unsubscribes.

A pub/sub system is called *self-stabilizing*, if started in an arbitrary state, it eventually begins to satisfy its specified behavior provided that no faults occur for a sufficient long time. A *fault* may lead to arbitrary perturbations of any variable stored in RAM as well as removed, manipulated, and inserted messages. Links may go down and come up, processes may crash and restart due to faults.

To guarantee persistence in spite of arbitrary memory faults, we assume that all algorithms used are stored in non-perturbable ROM. Additionally, we treat the reference to the root broker as well as the initial routing configuration as an intrinsic part of the respective algorithm itself and include them in ROM, too.

To maintain self-stabilization in case of crashed processes, the root broker R can be implemented in a self-stabilizing fashion using a root group [4].

While faults happen suddenly and may lead to abrupt changes in the broker topology, a reconfiguration is a *cooperative* process that is usually planned in advance and needs some time to take effect. More precisely, a *reconfiguration* is a change of the broker overlay topology, including leaf broker removals, additions, and link replacements, that can be delayed for a finite time.

Since reconfigurations affect several algorithm layers simultaneously, all actions carried out must be synchronized to meet the system's correctness requirements and to reach atomicity. A major challenge is to integrate the individual self-stabilizing algorithms of each layer into the whole reconfiguration process.

4 Layered Self-stabilization

Systems that are layered can be made self-stabilizing by making all layers individually self-stabilizing. This transparent stacking of self-stabilizing layers is a standard technique which is referred to as *fair composition* [2]. It is easy to combine self-stabilizing algorithms this way to create a new and more powerful self-stabilizing mechanism as long as no cyclic dependencies exist among the layers. Taking this approach, it is sensible to layer self-stabilizing routing in pub/sub systems on top of a broker topology that employs a self-stabilizing tree algorithm like the ones given in literature. However, this approach has its drawbacks because a reconfiguration on the broker overlay layer may be handled as a fault on the pub/sub layer when the routing table entries are not consistent with the new topology anymore. Additionally, most self-stabilizing tree algorithms impose a specific structure on the topology that is, for example, dependent on the IDs of the nodes. As a consequence, a topological reconfiguration of the self-stabilizing pub/sub system might result in a service interruption like missed notifications or control messages (subscriptions and unsubscriptions).

Our approach in the following is to realize a self-stabilizing overlay topology that maintains an arbitrary tree structure and to layer self-stabilizing routing on top of it in a way, such that reconfigurations of the overlay topology can be processed without service interruption. Two problems have to be tackled to solve this problem: (i) designing a self-stabilizing broker overlay topology that does not necessarily impose a certain structure on the resulting tree and (ii) coupling the self-stabilizing mechanisms on the overlay and the routing layer to allow for atomic topology switches without loss of messages.

Coloring Scheme. The coloring scheme synchronizes reconfigurations on the overlay layer with the routing layer. Therefore, selected data structures are marked with a *color* attribute. On the overlay topology layer this concerns the child and parent broker pointers ($\mathcal{C}$ and $\mathcal{P}$, respectively), while on the routing layer the routing entries are affected. To allow atomic switches between different colors, every broker maintains data structures for three different colors that can be accessed on both layers: the color c^{cur} that is currently used, the color c^{old}

that has been used last, and the color c^{new} that will be used when the color changes for the next time. These colors are rotated regularly. The reason why we need three different colors is due to the communication and processing delay in the network. If the value of c^{cur} becomes the value of c^{old}, for example, there may still be messages on the network that are colored with c^{old}. To be able to deliver these messages, the topology for c^{old} has to be kept long enough. For a better understanding, we assume in the following that the routing entries are stored in separate routing tables T for each color although a tag on each entry suffices in the implementation.

It is the task of the root broker R to regularly recolor all brokers in the tree. To accomplish this, a timeout runs on every broker that triggers different actions on R and on each broker $B \neq R$. On a timeout, R resets its timer, rotates its colors and subsequently initializes the child broker pointers $\mathcal{C}^{c^{\text{new}}}$ and the parent broker pointer $\mathcal{P}^{c^{\text{new}}}$ with the respective values colored with c^{cur} (which has been c^{new} before the timeout). The routing table $T^{c^{\text{new}}}$ is initialized with the initial routing configuration. Then, it disseminates the new color in a *recolor message* REC_{msg} to all child brokers stored in $\mathcal{C}^{c^{\text{cur}}}$, if they are still alive as indicated by a flag that was set when the child broker acknowledged the previous REC_{msg}. For every other broker $B \neq R$ a timeout is viewed as a fault and hence B tries to reconnect to the tree (as part of the self-stabilizing overlay topology). Reconfigurations are stored in REC_{msg} and handled as described later in a separate section. When B receives a recolor message, it resets its timer, replies with an *acknowledge message*, rotates its colors, initializes its pointers like R, and forwards the message to its child brokers. The broker accepts the recolor message only if it has been sent by the broker $\mathcal{P}^{c^{\text{new}}}$ points to and if the new color $m.c$ stored in the message is different from the color stored in c^{new}. This test is needed to detect cycles that may result from faults.

Self-Stabilizing Broker Overlay Topology. The self-stabilizing mechanism on the broker overlay network is based on timeouts regarding recolor messages as described above. Recolor messages are forwarded recursively down the tree, the last leaf broker receives the message at the latest after time $h \cdot \delta_{\max}$, where h is the height of the tree and $\delta_{\max}$ is the maximum delay for processing and sending a message to a child broker. As the tree may degenerate arbitrarily h can be at most equal to the maximum number of brokers η in the system (which we assume is known and stored in ROM). Given that the timeout on R occurs every time ξ, a timeout $\xi' = \xi + h \cdot \delta_{\max}$ is necessary on every broker B distinct from R, which is resetted everytime a new recolor message is received from its parent broker. When $B \neq R$ runs into a timeout, it took more than ξ' to receive the next recolor message after the last one. This can only be due to a fault, since forwarding a message from R to B cannot take more than $h \cdot \delta_{\max}$. In this case, B contacts R to rejoin the tree. There are many ways to find a new parent broker for B depending on the topology requirements. One is to look for an arbitrary broker that has less than b child brokers down the tree and use it as a new parent for a requesting broker. This way, the broker is integrated as a leaf into the tree and the degree of a the broker topology can be maintained. The broker

overlay is in a *correct state* if the parent and child broker relation between every broker in the system is consistent for the data structures colored with the values of c^{old} and c^{cur} at R and the tree that is defined by $\mathcal{P}^{c^{\text{old}}}$ and $\mathcal{C}^{c^{\text{old}}}$, and $\mathcal{P}^{c^{\text{cur}}}$ and $\mathcal{C}^{c^{\text{cur}}}$ respectively, is not partitioned. The value of $\mathcal{C}^{c^{\text{new}}}$ and $\mathcal{P}^{c^{\text{new}}}$ is treated differently as explained in the next subsection about reconfiguration.

Reconfiguration. Whenever a leaf broker wants to join or leave the overlay network or a link has to be replaced by another one, the topology of the broker network changes. When a reconfiguration should be implemented, the intended changes are sent to R, which collects them in the set $\mathfrak{R}$ and disseminates them in the next recolor message. Every broker that receives a recolor message carrying reconfiguration data that affects it, implements the change into its $\mathcal{P}^{c^{\text{new}}}$ and $\mathcal{C}^{c^{\text{new}}}$ pointers. The recolor message serves as a synchronizer to prevent race conditions when switching from one topology to another. Recolor messages are routed using $\mathcal{C}^{c^{\text{cur}}}$ of every broker B that receives a recolor message (where c^{cur} equals c^{new} before recoloring). Thus, reconfigurations take two recolor messages to become active: one to disseminate the reconfiguration and one to activate it.

As mentioned earlier, a change in the topology may imply a change in the routing tables on the pub/sub routing layer. As the routing tables are regularly rebuilt from an initial routing configuration the reconfiguration of the overlay topology can be incorporated by delaying the switch to the new topology in $\mathcal{P}^{c^{\text{new}}}$ and $\mathcal{C}^{c^{\text{new}}}$ long enough, such that they have been rebuilt completely.

Self-Stabilizing Routing. Recolor messages are used on the topology layer to trigger timeouts and coordinate reconfigurations. Therefore, three different topologies are held in form of colored parent/child pointers. On the routing layer, the color is used for two different purposes: (i) to rebuild the routing tables periodically and (ii) to avoid notification loss and duplicates.

It is necessary to periodically rebuild the routing tables as we assume that they can be perturbed arbitrarily. Therefore, we rely on the leasing mechanism described earlier [8]: clients regularly refresh their subscriptions and brokers use a second chance algorithm to remove stale entries from their routing tables. To incorporate reconfigurations into this mechanism, we demand that control messages are colored with c^{new}, while notifications are colored with c^{cur}. Notifications and control messages are then forwarded and applied to the routing tables $T^{c^{\text{cur}}}$ and $T^{c^{\text{new}}}$, respectively. Thereby, we ensure that notifications will be routed over the topology, the publishing broker belonged to at publishing time. This way, we prevent duplicates, i.e., notifications sent multiple times to the same broker. The second chance algorithm is implemented through rotating the colors and initializing $T^{c^{\text{new}}}$ with a legal initial routing configuration.

5 Summary

We presented an algorithm that allows self-stabilizing pub/sub systems to be reconfigured while maintaining service availability. To achieve this, we use the

color attribute to synchronize the self-stabilizing broker overlay and the self-stabilizing pub/sub routing layer. We connect the different layers such that it is possible to switch the topology atomically without losing or duplicating messages. The presumption is that reconfigurations can be delayed a bounded time before becoming active (e.g., before a broker is removed). We consider this "cooperative" behavior as the main difference between a fault and a reconfiguration. Our work is a necessary prerequisite to combine the self-stabilizing routing layer with an adaptive reconfiguration mechanism that runs on top of the pub/sub layer and issues reconfiguration stimuli that are implemented by the lower layers as described in this paper. Hence we come one step closer to fault-tolerant and adaptive publish/subscribe systems. However, the mechanism we described is not limited to self-stabilizing pub/sub. It is a general principle that can be used to realize reconfigurations in arbitrary layered self-stabilizing systems.

References

1. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
2. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
3. S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 4, Dec. 1997. Special Issue on Self-Stabilization.
4. S. Dolev and R. I. Kat. Hypertree for self-stabilizing peer-to-peer systems. In *Network Computing and Applications (NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 25–32, Washington, DC, USA, 2004. IEEE.
5. S. Ghosh, A. Gupta, T. Herman, and S. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium of Distributed Computing (PODC96)*, pages 45–54. ACM, ACM, 1996.
6. M. A. Jaeger and G. Mühl. Stochastic analysis and comparison of self-stabilizing routing algorithms for publish/subscribe systems. In G. F. Riley, R. Fujimoto, and H. Karatza, editors, *The 13th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2005)*, pages 471–479, Atlanta, Georgia, USA, Sept. 2005. IEEE Press.
7. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, Sept. 2002.
8. G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, and A. Ulbrich. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In J. C. Cunha and P. D. Medeiros, editors, *Proceedings of the 11th European Conference on Parallel Processing (Euro-Par 2005)*, volume 3648 of *Lecture Notes in Computer Science (LNCS)*, pages 664–674, Lisboa, Portugal, Aug. 2005. Springer.
9. V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.
10. Z. Shen and S. Tirthapura. Self-stabilizing routing in publish-subscribe systems. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, pages 92–97, Edinburgh, Scotland, UK, May 2004. IEE.