Self-stabilizing Publish/Subscribe Systems: Algorithms and Evaluation

Gero Mühl[1,⋆], Michael A. Jaeger[1,⋆⋆], Klaus Herrmann[1,⋆], Torben Weis[1,⋆⋆], Andreas Ulbrich[1,⋆], and Ludger Fiege[2]

[1] TU Berlin, EN6, Einsteinufer 17, 10587 Berlin, Germany
{g_muehl,michael.jaeger,klaus.herrmann}@acm.org,
{weis,ulbi}@ivs.tu-berlin.de
[2] TU Darmstadt, Wilhelminenstraße 7, 64283 Darmstadt, Germany
fiege@acm.org

Abstract. Most research in the area of publish/subscribe systems has not considered fault-tolerance as a central design issues. However, faults do obviously occur and masking all faults is at least expensive if not impossible. A potential alternative (or sensible supplementation) to fault masking is self-stabilization which allows a system to recover from arbitrary transient faults such as memory perturbations, communication errors, and process crashes with subsequent recoveries.
In this paper we discuss how publish/subscribe systems can be made self-stabilizing by using self-stabilizing content-based routing. When the time between consecutive faults is long enough, corrupted parts of the routing tables are removed, while correct parts are refreshed in time, and missing parts are inserted. To judge the efficiency of self-stabilizing content-based routing, we compare it to flooding, which is the naïve implementation of a self-stabilizing publish/subscribe system. We show that our approach is superior to flooding for a large range of practical settings.

1 Introduction

In many applications, independently created components have to be integrated into complex information systems. Especially in large-scale distributed applications, a loosely-coupled event-based style of communication has many advantages. It allows the clear separation of communication from computation and eases the integration of autonomous, heterogeneous components.

In publish/subscribe systems individual processing entities, which we call *clients*, can *publish* information without specifying a particular destination. Similarly, clients can express their interest in receiving certain types of information by *subscribing*. Clients can be *producers* and *consumers* at the same time. Information is encapsulated in *notifications* and the *notification service* is responsible for notifying each consumer about all occurrences of notifications which match one of its active subscriptions.

⋆ Funded by Deutsche Telekom.
⋆⋆ Funded by Deutsche Telekom Stiftung.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 664–674, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Many research prototypes of notifications services exist including Siena [2], Gryphon [9], Hermes [10], and Rebeca [7]. The Java Message Service (JMS) [12] and the Corba Notification Service [8] are two prominent examples of industrial specifications of notification services. However, in most research prototypes and industrial specifications fault-tolerance has not been a central design issue as the focus was mostly put on the efficiency of routing. Obviously, faults do occur and considering all kinds of faults when implementing fault masking is at least expensive if not impossible.

A potential alternative (or sensible supplementation) to fault masking is *self-stabilization*, a concept introduced by Dijkstra [3] in 1974. He defined a system as being self-stabilizing if "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". In contrast to that, a system which is not self-stabilizing may stay in illegitimate states forever leading to a permanent failure of the system. Self-stabilization models the ability of a system to recover from arbitrary transient faults within a finite time without any intervention from the outside. If the time between consecutive faults is long enough, the system will start to work correctly again. Transient faults include temporary network link failures resulting in message duplication, loss, corruption, or insertion, arbitrary sequences of process crashes and subsequent recoveries, and arbitrary perturbations of the data structures of any fraction of the processes. The program code running at the nodes and inputs from the outside, however, cannot be corrupted. Dolev [4] gives a comprehensive discussion of self-stabilization.

The remainder of this paper is structured as follows: In Sect. 2 we introduce the notion of self-stabilizing publish/subscribe systems. In Sect. 3, we show how specific routing algorithms can be made self-stabilizing. Sect. 4 presents our comparison of self-stabilizing identity-based routing with flooding. Sect. 5 presents some related work. We close with conclusions and give an outlook in Sec. 6.

2 Self-stabilizing Publish/Subscribe Systems

In previous work, Mühl, Fiege, and Gärtner presented a formalization of publish/subscribe systems as a requirement specification [5, 7] consisting of safety and liveness properties. Due to spatial restrictions, we only give an informal definition of our specification here:

Definition 1. *A* publish/subscribe system *is a system satisfying the following requirements:*

1. *Safety Property*
   (a) *A notification is only delivered to a client at most once.*
   (b) *A client only receives notifications that have previously been published.*
   (c) *A client only receives notifications it is subscribed for.*
2. *Liveness Property: When a client subscribes to a filter and does not issue an unsubscription for this filter, then, from some time on, every notification that is published thereafter and matches the filter will be delivered to the subscribing client.*

Content-based routing is one possibility to implement a distributed notification service. In this case, the notification service is realized by a set of *brokers* forming an overlay network. Here, we restrict ourselves to acyclic connected topologies. This restriction can be circumvented, e.g. by running a spanning tree algorithm on the original (potentially cyclic) topology. Each broker B communicates with its neighbor brokers N_B using *asynchronous message passing* and with its mutually exclusive set of local clients L_B using *local synchronous procedure calls*. The private *routing table* T_B of a broker B determines to which neighbors and local clients broker B forwards a notification that it processes. Each *routing entry* is a pair (F, D) consisting of a filter F having a unique id $id(F)$ and a *destination* $D \in N_B \cup L_B$. A broker sends a notification that it processes to all destinations for which a matching filter exists. However, if a notification is received from a neighbor broker, it is not sent back to this broker.

The routing table determines the current *routing configuration* of a publish/subscribe system. A *routing algorithm* starts from an eligible initial routing configuration and subsequently adapts it. To achieve this, *update messages* are propagated through the broker network when clients issue new or cancel existing subscriptions. Intuitively, a routing algorithm is *valid* if it adapts the routing configuration such that the resulting system satisfies the safety and the liveness property of Def. 1. Several content-based routing algorithms are known, including simple, identity-based, covering-based, and merging-based routing [7]. These algorithms exist in a peer-to-peer and in a hierarchical variant [1].

Definition 1 requires that the system is correct, i.e. exhibits the desired functionality at its interface, under all circumstances. Thus, all occurring faults would have to be masked. Provided that a temporary failure of the system can be accepted, making a system self-stabilizing is an attractive alternative to fault masking. However, it is in general impossible under the fault assumption of self-stabilization to require *any* property that prohibits certain states, i.e. safety properties. For example, the system could deliver a notification n to a client X although X has no active subscription matching n because a fault corrupted the state of the system such that that it "thinks" that X subscribed to n. Therefore, we require that a self-stabilizing publish/subscribe system satisfies the safety property of Def. 1 only eventually. This ensures that the system starting from any state will eventually satisfy the actual safety property and continue to do so if no faults occur. The liveness property of Def. 1 can be left unchanged. This leads to the following definition:

Definition 2. *A* self-stabilizing publish/subscribe system *is a system satisfying the following requirements:*

1. *Eventual Safety Property: Starting from any state, it eventually satisfies the safety property of Def. 1.*
2. *Liveness Property: Starting from any state, it satisfies the liveness property of Def. 1.*

In the following section, we discuss how self-stabilizing publish/subscribe systems can be realized using self-stabilizing content-based routing algorithms.

3 Self-stabilizing Content-Based Routing

Under the fault assumption of self-stabilization, the routing configuration can arbitrarily be corrupted by transient faults. Therefore, the routing algorithm must ensure that corrupted routing entries are corrected or deleted from the routing table and that missing routing entries are inserted into the routing table.

For spatial reasons we assume in this paper that each broker stores the information about its neighbors in its ROM. This ensures that this information cannot be corrupted. If it would be stored in RAM or on harddisk, it could also be corrupted by a fault. In this case, we would have to layer self-stabilizing content-based routing on top of a self-stabilizing spanning tree algorithm. Layered composition of self-stabilizing algorithms is a standard technique which is easy to realize when the individual layers have no cyclic state dependencies [4]. In this case, the stabilization time would be bounded by the sum of the stabilization times of the individual layers.

3.1 Basic Idea

The basic idea for making content-based routing self-stabilizing is that routing entries are only *leased*. To keep a routing entry, it must be renewed before the *leasing period* π has expired. If a routing entry is not renewed in time, it is removed from the routing table. Interestingly, this approach does not only allow the publish/subscribe system to recover from internal faults but also from certain external faults. For example, if a client crashes, its subscriptions are automatically removed after their leases have expired.

To support leasing of routing table entries, we use a *second chance* algorithm. Routing entries are extended by a *flag* that can only take the two values 1 and 0. Before a routing entry is (re)inserted into the routing table, all existing routing entries whose filter has the same *id* (as the *id* of the filter of the routing entry to be inserted) are removed from the routing table. This is necessary as the *id*s of the routing entries can be corrupted, too. We assume that the clock of a broker can only take values between 0 and $\pi - 1$ to ensures that if the clock is corrupted, it can diverge from the correct clock value by at most π. When its clock overruns, a broker deletes all routing entries whose flag has the value 0 from the routing table and sets the flag of all remaining routing entries to 0 thereafter (new subscriptions have the flag set to 1 initially). Hence, it must be ensured that an entry is renewed once in π to prevent its expiry. On the other hand, it is guaranteed that an entry which is not renewed will be removed from the routing table after at most 2π.

The renewal of routing entries originates at the clients. To maintain its subscriptions without interruption, a client must renew the lease for each of its subscriptions by "resubscribing" to the respective filter once in a *refresh period* ρ. Resubscribing to a filter is done in the same way as subscribing. In general, π must be chosen to be greater than ρ due to varying link delays. The *link delay* δ is the amount of time needed to forward a message over a communication link and to process this message at the receiving broker. In our model, it is considered

a fault when δ is not in the range between $\delta_{\min}$ and $\delta_{\max}$. It is important to note, that assuming an upper bound for the link delay is a necessary precondition for realizing self-stabilization.

3.2 Flooding

The naïve implementation of a self-stabilizing publish/subscribe system is *flooding*: When a broker receives a notification from a local client, the broker forwards the notification to all neighbor brokers. When it receives a notification from a neighbor broker, the notification is forwarded to all other neighbor brokers. Additionally, each processed notification is delivered to all local clients with a matching subscription. Flooding only requires a broker to keep state about the subscriptions of its local clients. Therefore, errors in this state can be corrected locally by forcing clients to renew their subscriptions once in a leasing period. This means that $\rho = \pi$. The main advantage of this scheme is that a coordination among neighboring brokers is not necessary. Hence, no additional network traffic is generated. Additionally, new subscriptions become active immediately. While a corrupted or erroneously inserted subscription survives at most 2π in a routing table and a missing subscription is reinserted after at most π, an erroneously inserted or corrupted notification disappears from the network after at most $d \cdot \delta_{\max}$ where d is the *network diameter*, i.e. the length of the longest path a message can take in the broker network. Hence, for flooding, the *stabilization time* Δ, i.e. the time it takes for the system to reach a legitimate state starting from an arbitrary state, equals $\max\{2\pi, d \cdot \delta_{\max}\}$.

3.3 Simple Routing

The solution for flooding can be extended to simple routing. *Simple routing* treats each subscription independently of other subscriptions. A (un)subscription is inserted into (removed from) the routing table and flooded into the broker network. If a broker receives a (un)subscription from a local client, it is forwarded to all neighbor brokers. If it was received from a neighbor broker, it is forwarded to all other neighbor brokers. Thus, simple routing is idempotent to resubscriptions and a subscription is redistributed through the broker network when it is renewed by the client. Note that here subscriptions become active only gradually.

A critical issue is that the timing assumptions must allow the clients to renew their leases everywhere in the network before they expire. How large must π be with respect to ρ in this case? To answer this question, consider two brokers B and B' connected by the longest path a message can take in the broker network. This situation is illustrated in Fig. 1. Assume a local client X of B leases a routing table entry of B at time t_0 and renews this lease at time $t_1 = t_0 + \rho$. X's lease causes other leases to be granted all along the path to broker B'. Considering the best and worst cases of the link delay, the first lease reaches B' at time $a_0 = t_0 + d \cdot \delta_{\min}$ in the best case and the lease renewal reaches B' at time $a_1 = t_1 + d \cdot \delta_{\max}$ in the worst case. If X refreshes its leases after ρ time and if network delays are unfavorable, two lease renewals will arrive at B' within at

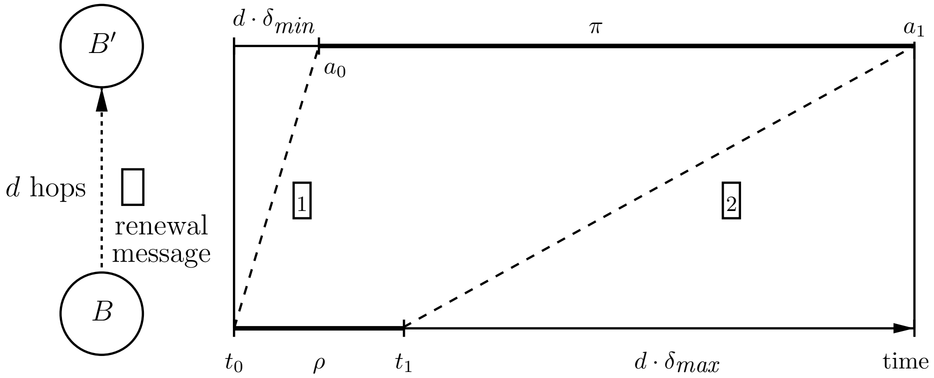

Fig. 1. Deriving the Minimum Leasing Time.

most $a_1 - a_0$. Hence, $\pi > a_1 - a_0$ must hold to ensure that the entry is renewed in time. Thus, we get $\pi > \rho + d \cdot (\delta_{\max} - \delta_{\min})$.

The stabilization time Δ depends on the value of π. Since corrupted or erroneously inserted messages can contaminate the network, a delay of $d \cdot \delta_{\max}$ must be assumed before their processing is finished. After at most 2π, their effects will be removed everywhere. Overall, the stabilization time sums up to $\Delta = d \cdot (\delta_{\max} - \delta_{\min}) + 2\pi$. For example, assume that $d = 10$, $\delta_{\max} = 25\,ms$, and $\delta_{\min} = 5\,ms$. To guarantee a stabilization time of $\Delta = 30\,s$, $\pi = 14.9\,s$ and thus $\rho = 14.7\,s$ follows. There is a tradeoff between π and ρ. To have low message overhead, ρ should be as large as possible. However, this implies a large value of π, but π should be as small as possible to facilitate fast recovery.

3.4 Advanced Routing Algorithms

The situation is more complicated if advanced content-based routing algorithms such as identity-based, covering-based, or merging-based routing are applied. Contrary to flooding and simple routing these algorithms are – at least the versions presented so far – not idempotent with respect to resubscriptions. However, they can be made idempotent with some minor changes. Note that the maximum stabilization time Δ is not affected by whether an advanced routing algorithm or simple routing is applied because in the worst case a filter will nevertheless travel all along the longest path in the network.

Consider identity-based routing (for more details we refer to [7]). When a broker B processes a new or canceled subscription F from destination D, it counts the number d of destinations $D' \neq D$ for which a subscription matching the same set of notifications exists in T_B. Depending on the value of d, F is forwarded differently. If $d = 0$, F is forwarded to all neighbors if $D \in L_B$ and to all neighbors except D if $D \in N_B$. If $d = 1$ and $D' \in N_B$, F is forwarded only to D'. If $d = 1$ and $D' \in L_B$ or if $d \geq 2$, F is not forwarded at all. This scheme is not idempotent to resubscriptions because if $d \geq 2$ and one of the identical subscriptions is renewed at B, none of those subscriptions will be forwarded.

This can be circumvented if B takes only those subscriptions into account when calculating d whose flag is 1. In this case, in each leasing period that subscription of the identical subscriptions which is renewed first after the broker has run the second chance algorithm, is forwarded ensuring correct forwarding.

Covering-based routing can also be made self-stabilizing. In this case, only routing entries with flag 1 are taken into account when looking for identical subscriptions. However, when looking for subscriptions that really cover a given subscription (i.e. match a real superset of notifications), additionally also those routing entries with flag 0 are considered. This is to avoid sending covered subscriptions unnecessarily to neighbors because they are refreshed before a covering subscription is refreshed. To make merging-based routing self-stabilizing, the refreshing of merged filters must additionally be ensured.

3.5 Discussion

The values of π and ρ depend on the delay of the links in the network. So far, we assumed that these values are fixed and equal for every broker in the system. In many scenarios, link delays vary a lot such that it could be advantageous to incorporate this property into the algorithm. We assume that the value of link delay stored at every adjacent broker can not be corrupted (i.e. it is stored in ROM). The values of π and ρ has then to be calculated individually for every subscription, depending on where the publishers are. Additionally, π and ρ have to be refreshed the same way as described previously for subscriptions. Advertisements that are sent periodically by the publishers could be used for this purpose. Taking this approach, the broker algorithm can take advantage of faster links and stabilizes subtrees of the broker topology faster if the links allow for this. The application of leasing is a common way to keep soft states. This technique is used in many protocols and algorithms such as the Routing Information Protocol (RIP, RFC2453) and Directed Diffusion [6].

4 Simulation

We carried out a discrete event simulation to compare self-stabilizing content-based routing to flooding with respect to their message complexity. Before we discuss the results, we describe the setup of the experiments.

4.1 Setup

We consider a broker hierarchy being a completely filled 3-ary tree with 5 levels. Hence, the hierarchy consists of 121 brokers of which 81 are leaf brokers. Since we use a tree for routing, this implies a total number of 120 communication links. We use hierarchical routing but similar results can be obtained for peer-to-peer routing, too. With hierarchical routing, subscriptions are only propagated from the broker to which the subscribing client is connected towards the root broker. This suffices because every notification is routed through the root broker. Hence,

control messages travel over at most 4 links. We use identity-based routing and consider 1000 different filter classes (e.g. stocks) to which clients can subscribe.

Subscribers only attach to leaf brokers. Results for scenarios where clients can attach to every broker in the hierarchy can be derived similarly. Instead of dealing with clients directly, we assume independent arrivals of new subscriptions with exponentially distributed interarrival times and an expected time of λ^{-1} between consecutive arrivals. When a new subscription arrives, it is assigned randomly to one of the leaf brokers and one of the filter classes is randomly chosen. The lifetime of individual subscriptions is exponentially distributed with an expected lifetime of μ^{-1}. Each notification is published at a randomly chosen leaf broker. Hence, notifications travel over at most 8 links. The corresponding filter class is also chosen randomly. The interarrival times between consecutive publications are exponentially distributed with an expected delay of ω^{-1}. We assume a constant delay in the overlay network of $\delta = 25\,ms$ including the communication and the processing delay caused by the receiving broker.

To illustrate the effects of changing the parameters, we considered two possible values for some of the system parameters: For each of the 1000 filter classes, a publication is expected every $1\,s$ ($10\,s$), i.e. $\omega_1 = 1000\,s^{-1}$ ($\omega_2 = 100\,s^{-1}$). The expected subscription lifetime is $600\,s$ ($60\,s$), i.e. $\mu_1 = (600\,s)^{-1}$ ($\mu_2 = (60\,s)^{-1}$). Each client refreshes its subscriptions once in $60\,s$ ($600\,s$), i.e. a refresh period of $\rho_1 = 60\,s$ ($\rho_2 = 600\,s$). Since $d = 8$ in our scenario, the leasing period is $\pi_1 = 60.2\,s$ ($\pi_2 = 600.2\,s$) for ρ_1 (ρ_2). Hence, a subscription will on average be refreshed 10 (100) times before it is canceled by the subscribing client if $\mu = (600\,s)^{-1}$. The resulting stabilization time is $\Delta_1 = 120.6\,s$ ($\Delta_2 = 1200.6\,s$).

We are interested in how the system behaves in equilibrium for different numbers of active subscriptions N. In equilibrium, $dN/dt = 0$ where $dN/dt = \lambda - \mu \cdot N(t)$, implying $N = \lambda/\mu$. Thus, if N and μ is given, λ can be determined. If the system was started with no active subscriptions, we would have to wait until the system approximately reached equilibrium before we begin the measurements. However, in our scenario it is possible to start the system right in the equilibrium. At time 0, we create N subscriptions. For each of these subscriptions, we determine how long it will live, for which filter class it is, and at which leaf broker it is allocated. Since we use an exponential distribution for the lifetime, this approach is feasible because the exponential distribution is memoryless.

4.2 Results

The results of our simulation are depicted in Fig. 2. Note that the right plot is a magnification of the most interesting part of the left plot. $b_{s1/2}$ is the notification bandwidth saved if filtering is applied instead of flooding. The figure shows b_{s1} and b_{s2} which correspond to the publication rate ω_1 and ω_2, respectively. Because b_s linearly depends on ω, a decrease of ω by a factor of 10 leads to 10 times less saving of notification bandwidth. If there are no subscriptions in the system, $b_{s1} = 116,000\,s^{-1}$ and $b_{s2} = 11,600\,s^{-1}$, respectively. These numbers are $4,000\,s^{-1}$ and $400\,s^{-1}$ less than the overall number of notifications

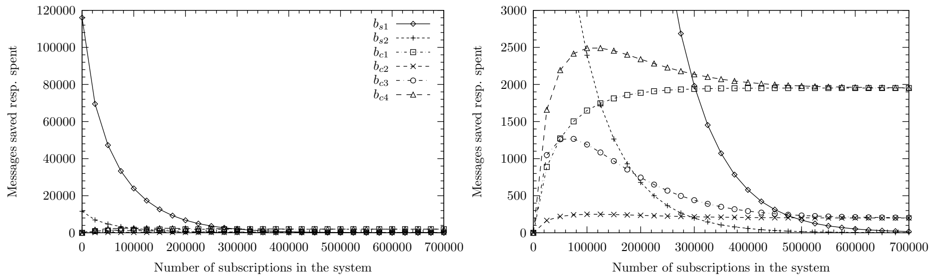

Fig. 2. Notification bandwidth saved by doing filtering instead of flooding ($b_{s1} : \omega_1 = 1000\, s^{-1}, b_{s2} : \omega_2 = 100\, s^{-1}$) and control traffic caused by filtering and leasing ($b_{c1}, b_{c4}, :$ $\rho_1 = 60\, s$, $b_{c2}, b_{c3}, : \rho_2 = 600\, s$, $b_{c1}, b_{c2} : \mu_1 = (600\, s)^{-1}$, $b_{c3}, b_{c4} : \mu_2 = (60\, s)^{-1}$).

published per second. This is because with hierarchical routing, a notification is always propagated to the root broker. The control traffic b_c is caused by subscribing, refreshing, and unsubscribing clients. It only arises if filtering is used. The figure shows b_{c1}, b_{c2}, b_{c3}, and b_{c4} which result from the different combinations of μ and ρ. The value to which b_c converges for large numbers of subscriptions, mainly depends on the refresh period ρ. Thus, b_{c1} and b_{c3} converge to $120,000/\rho_1 = 2,000 s^{-1}$, while b_{c2} and b_{c4} converge to $120,000/\rho_2 = 200 s^{-1}$. The evolution of b_c for numbers of subscriptions in the range between 0 and $200,000$ is largely influenced by the value of μ. A small μ such as μ_2 leads to a hump (cf. b_{c3} and b_{c4} in Fig. 2). Filtering saves bandwidth compared to flooding if b_s exceeds b_c. The points where the curve of the respective variants of b_s and b_c intersects are important: If the number of subscriptions is smaller than at the intersection point, filtering is superior, while for larger numbers flooding is better. For example, the curves of b_{s1} and b_{c1} intersect for about $300,000$ subscriptions. Thus, filtering is superior for less than $300,000$ subscriptions, while flooding is superior for more than $300,000$ subscriptions. Since we consider 8 scenarios, we have 8 intersection points in Fig. 2.

The results gained through the simulation show, that applying self-stabilizing filtering makes sense if the average number of subscriptions in the system does not grow beyond a certain point. However, it is important to note, that all assumptions taken for the simulation depict worst-case scenarios. For example, the equal distribution of subscriptions to leaf brokers is disadvantageous for filtering. If there was locality in the interests of the clients, filtering would always save a portion of the notification traffic regardless how large the number of subscriptions grows [7] and the control traffic would also be smaller. In such scenarios, filtering can be superior to flooding for all numbers of subscriptions.

5 Related Work

Many self-stabilizing algorithms have been proposed for various kinds of scenarios whilst there are only a few contributions that cover publish/subscribe

systems. In this area self-stabilization was first considered by Mühl [7]. This work was used as the basis for this paper. Recently, Shen and Tirthapura [11] presented an alternative approach for self-stabilizing content-based routing. In their approach, all pairs of neighboring brokers periodically exchange sketches of those parts of their routing tables concerning their other neighbors to detect corruption. The sketches that are exchanged are lossy because they are based on bloom filters (which are a generalization of hash functions). However, due to the information loss, it is not guaranteed that an existing corruption is detected deterministically. Hence, the algorithm is not self-stabilizing in the usual sense. Moreover, although generally all data structures can be corrupted arbitrarily, the authors' algorithm computes the bloom filters incrementally. Thus, once a bloom filter is corrupted, it may never be corrected. Furthermore, in their algorithm, clients do not renew their subscriptions. Without this, corrupted routing entries regarding local clients are never corrected. Finally, their algorithm is restricted to simple routing in its current form.

6 Conclusions and Outlook

To make publish/subscribe systems self-stabilizing, we applied a leasing mechanism ensuring that the routing tables are always refreshed in time provided that no faults occur. When faults do occur, the leasing mechanism ensures (a) that corrupted parts of routing tables are either corrected or removed and (b) that missing part are inserted. This way, routing tables recover. We described how flooding and simple routing can be made self-stabilizing. In both cases, we calculated the maximum stabilization time, i.e. the time the system needs to recover from an error. We also described how the stabilization time depends on the leasing period and how the refresh period must be chosen to ensure that in a correct system no routing entries expire. Furthermore, we sketched how advanced routing algorithms can be made self-stabilizing. Our contributions in this paper enable the designers of publish/subscribe systems to render their system self-stabilizing. Therefore, designers and implementers need not consider explicit fault management mechanisms if fault masking is not an issue.

Using a simulation we tested the effectiveness of our approach in an example scenario and showed, that it depends on the number of subscriptions in the system. In future work, it would be interesting to take an analytical approach to judge the proposed algorithms without employing simulations.

In this paper, we assumed for spatial reasons that the broker topology is statically stored in ROM. Currently, we work on an algorithm for a self-stabilizing broker topology which ensures the correct behavior of the system even if nodes or links are added or removed from the broker topology. Besides this, we are investigating self-organizing and self-optimizing algorithms for managing the broker topology. These management algorithms decide on which hosts brokers are started and to which neighbor brokers a broker connects.

References

1. A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.
2. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
3. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
4. S. Dolev. *Self-Stabilization*. MIT Press, 2000.
5. L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, 2003.
6. C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking (TON)*, 11(1):2–16, 2003.
7. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002. http://elib.tu-darmstadt.de/diss/000274/.
8. OMG. CORBA notification service, version 1.0.1. OMG Document formal/2002-08-04, 2002.
9. L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, volume 1795 of *LNCS*, pages 185–207. Springer-Verlag, 2000.
10. P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, July 2002.
11. Z. Shen and S. Tirthapura. Self-stabilizing routing in publish-subscribe systems. In *3rd International Workshop on Distributed Event-Based Systems (DEBS 2004)*, Edinburgh, Scotland, UK, May 2004.
12. Sun Microsystems, Inc. Java Message Service (JMS) Specification 1.1, 2002.