Reconfiguring Publish/Subscribe Overlay Topologies

Helge Parzyjegla[*] Gero Mühl[†] Michael A. Jaeger[‡]
Communication and Operating Systems Group
Berlin University of Technology
Einsteinufer 17, 10587 Berlin, Germany
{parzyjegla,g_muehl,michael.jaeger}@acm.org

Abstract

Distributed content-based publish/subscribe systems are usually implemented by a set of brokers forming an overlay network. Most existing publish/subscribe middleware assumes that the overlay topology is static. Those that consider reconfiguration, exchange a single link that is torn down by another link that comes up. However, they do not guarantee that no notifications are lost or duplicated nor do they ensure any message ordering policy.

In this paper, we discuss the dynamic reconfiguration of publish/subscribe systems which are built on content-based routing. We present algorithms that allow for reconfigurations without notification loss or duplication that can also ensure FIFO-publisher and causal ordering. Moreover, the efficiency of reconfigurations is increased by limiting their effects to those parts of the network which are directly affected by the reconfiguration.

Keywords: Publish/Subscribe, Content-Based Routing, Topology Reconfigurations, Dynamic Overlay Networks

1 Introduction

Distributed content-based publish/subscribe systems provide asynchronous, implicit, and flexible group communication well suited for dynamic environments such as wide area networks with arbitrarily joining and leaving end nodes. By publishing notifications and subscribing to notifications of interest, the publish/subscribe middleware enables application developers to abstract from network and communication details. Developers can thus focus on the information to exchange and the application itself.

State-of-the-art distributed publish/subscribe systems are usually implemented by a set of brokers forming an overlay topology. However, most systems restrict themselves to static broker topologies or at the very least restrict the dynamic evolution of the topology. In consequence, they do not support dynamic environments adequately. Often it is desirable (e.g., for balancing the broker load and for reducing the overall network traffic), sometimes even necessary (e.g., to extend the system), to modify the broker topology. This may include starting new brokers, shutting down running brokers, and rearranging links inside the broker network. Thus, it is sensible to require that the publish/subscribe overlay topology is capable to dynamically reconfigure itself.

Approaches in literature that consider reconfiguration by repeatedly exchanging a single link [1, 2, 5] fail to guarantee the completeness or ordering of notifications during reconfigurations. In this paper, we discuss algorithms which allow reconfigurations without notification loss or duplication and that ensure FIFO-publisher and causal ordering.

The remainder of this paper is structured as follows: Section 2 discusses the reconfiguration algorithms. Section 3 presents an evaluation of the reconfiguration algorithms based on simulations. In Sect. 4 we discuss related work. Finally, we present our conclusions.

2 Reconfiguration Algorithms

We start with the description of three elementary types of reconfigurations that can be used to carry out any other complex topology reconfiguration. Then, we show how they can be used to construct arbitrarily complex reconfigurations. We assume an acyclic overlay topology where message ordering on the links is FIFO and reconfigurations are controlled, i.e., the removal of a link or broker can be delayed for a finite time.

2.1 Elementary Types of Reconfigurations

We consider the following actions as basic types of reconfigurations:

[*]Funded by Deutsche Forschungsgemeinschaft.
[†]Funded by Deutsche Telekom.
[‡]Funded by Deutsche Telekom Stiftung.

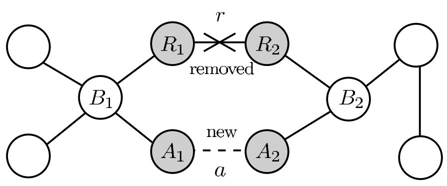

Figure 1. Exchanging a link in an acyclic topology.

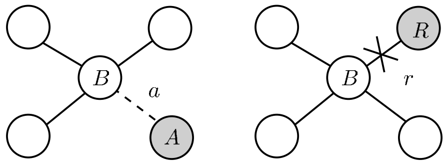

Figure 2. Adding (left subfigure) and removing (right subfigure) of a leaf broker.

Link Exchange. An existing link is removed from the topology and a new link is added instead that reconnects the two resulting partitions again (see Fig. 1).

Adding a Leaf Broker. A broker is added as a leaf broker to the topology (see left part of Fig. 2).

Removing a Leaf Broker. A leaf broker is removed from the topology (see right part of Fig. 2).

With a sequence of these three basic reconfigurations it is possible to carry out more complex reconfigurations. For example, inserting a new broker B between two connected brokers B_1 and B_2 can be done by first adding B as a leaf of B_1 and then exchanging the link between the B_1 and B_2 by a link connecting B with B_2. In general, inserting a new broker of degree n requires connecting the broker as a leaf and $n - 1$ additional link exchanges.

2.2 Link Exchange

Exchanging a link does not only mean to substitute the removed link r with the added link a, but also includes updating the brokers' routing tables to reflect the change in the network topology. This affects all brokers that lie on the *reconfiguration cycle*, which results from inserting a into the present broker tree and includes a, r, and all the links on the path from A_1 to R_1 and A_2 to R_2, respectively (cf. Fig. 1). These brokers have to ensure, that all notifications previously routed over r are now forwarded towards and over the new link a instead.

2.2.1 The Uncoordinated Approach

In the naïve approach, R_1 and R_2 act as if they had received unsubscriptions for each of their routing entries associated with r before tearing down r. Simultaneously, A_1 and A_2 issue a subscription over a for each of their current routing entries. Since subscriptions and unsubscriptions are processed and forwarded concurrently without a dedicated ordering, we call this an uncoordinated reconfiguration. In literature it is also referred to as the *strawman approach* [5].

There are several problems related to race conditions that may cause undesired effects when omitting coordination. Notifications might get lost, when a broker B on the reconfiguration cycle (e.g., B_1) processes an unsubcription originated at r, before the corresponding resubscription arrived from a. Furthermore, if B then removes its last corresponding routing entry, depending on the routing algorithm, the unsubscription might also be forwarded to neighboring brokers, that are not part of the reconfiguration cycle. Hence, the reconfiguration process entails unintentional global effects, too. Besides the loss of notifications and global effects, also duplicates can occur and the ordering of notifications might get mixed up: while the original notification passes the old link before r is torn down, a copy of the same notification might be in transit to another subscriber on the route to a. When this copy encounters one of a's endpoints later, it is also forwarded over the new link, if a is already set up at this time. In this case, the copy might reach parts of the network before the original notification or even afore published notifications arrive.

2.2.2 The Coordinated Approach

To face these problems caused by concurrency, we coordinate actions and split the process into two phases. In the first phase, the new link a is set up and all routing entries are established along the reconfiguration cycle such that the old link r can be substituted. In the second phase, r is torn down and superfluous entries pointing directly or indirectly to r are removed. We discuss both phases in detail below focussing on minimizing the overhead and limiting effects to the reconfiguration cycle. Afterwards, we add measures to ensure completeness and ordering of notifications and subscriptions. Finally, we consider concurrent reconfigurations. The algorithms' pseudocode can be found in [4].

Phase 1: Establishing the New Link. Since the topology is acyclic, the link r connects two partitions of the network. Exchanging r by a thus means that all notifications that where forwarded over r have to be forwarded over a now. In consequence, the routing tables at A_1 and A_2 have to contain equivalent or the same entries for a as R_1 and R_2 did for r. To accomplish this, R_1 and R_2 create subscriptions for all entries associated with r and send them (i.e., out-of-band) to A_1 and A_2, respectively. A_1 and A_2 process the subscriptions as if they came from the opposite side over the new link and forward them accordingly.

In the following, we illustrate the algorithms only for the left side of the reconfiguration cycle of the example in Fig. 3 as actions for the right side are analogous. For the identifiers we also refer to this figure.

Two scenarios are possible. In the first, presumably representing the majority of all cases, subscription s sent by broker R_1 to broker A_1 is already active on the path between these two brokers. This means, that s is already incorporated into the routing tables of the brokers on the path from R_1 to A_1 and was also sent to neighboring brokers (N_1 and N_2 in Fig. 3) if necessary. However, the routing entries established route every matching notification on the path to A_1 towards the old link r that will be removed. When A_1 issues s received from R_1, it will only be forwarded along the path from A_1 to R_1 as part of the regular filter forwarding mechanism of the routing algorithm used. The subscription will not leave the reconfiguration cycle, since the original copy coming from R_1 was already sent to every other broker on the path as assumed above.

In the second scenario, s is not yet active on the left side of the reconfiguration cycle, since it is still in transit towards a. In this case, the copy of s sent to A_1 is sent along the cycle in the opposite direction heading towards the original s. Both copies might also be forwarded to other neighboring brokers (like N_1 and N_2) to ensure the subscription's dissemination in the entire network according to the routing algorithm used and the state of the routing tables. After both copies have met each other on the reconfiguration cycle, they strictly follow the reverse path of the other one which they will not leave in the following, as the other copy of s has already established all routing entries that could lead away from the reconfiguration cycle.

Although we assume a simple subscription forwarding scheme when describing the algorithms, they are still applicable if more sophisticated mechanisms such as identity-based or covering-based routing are used. In this case subscriptions transferred from r to a might not need to be sent along the whole path, if an identical or a covering subscription has already established the routing entries before.

The end of phase 1 is marked by a special *separator message* which is sent by R_1 to A_1. It signals that all subscriptions from R_1 have been sent to A_1. After receiving the message, A_1 forwards it along the reconfiguration cycle as depicted in Fig. 3 until it reaches A_2.

Phase 2: Tearing Down the Old Link. When a separator message reaches its originating broker R_1 again, it is guaranteed, due to the FIFO property of the links, that all subscriptions at R_1 have been completely transferred to A_1. This means that superflous routing entries pointing to r can now be removed. Therefore, R_1 issues an unsubscription for every routing table entry associated with r, which is then processed in the regular way.

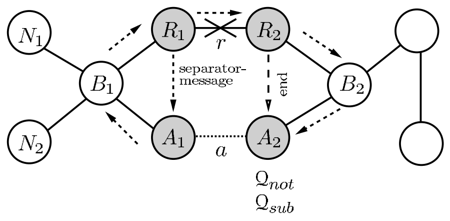

Figure 3. Separator messages and message queues.

In contrast to the strawman approach, it is impossible that a broker removes the last routing entry for a subscription during this phase of the reconfiguration, since the corresponding subscription from R_1's side of r had been issued in phase 1 and is already present at each node on the path from R_1 to A_1. Hence, the unsubscription is only forwarded along the reconfiguration cycle towards A_1 and not propagated to any other neighboring broker causing unintended global effects this way.

When R_1 and R_2 have each received both separator messages (their own and the one from the other side forwarded over r), the new link and its routing entries have been established on the whole reconfiguration cycle. In consequence, r can be torn down and the reconfiguration is finished.

Notification Loss. In the first phase, new subscriptions are established to ensure that notifications previously routed from one side of r to the other will now be routed over a. As no old subscription is removed, notifications will not get lost. When all new subscriptions are established, the old ones pointing over r will be removed. As the subscriptions for a have already been established in the phase before, no notifications will get lost in this phase, too, as there is always at least one path from every publisher to each subscriber. On the downside, the approach taken may produce duplicates, which we will tackle in the following.

Avoiding Duplicates. As subscriptions might get duplicated in the first two phases when r and a are active, we introduce a *color* attribute for brokers and messages (i.e., control messages and notifications), which is only used during reconfiguration. According to the color, we can sort out which messages have to be forwarded over r and which have to be sent over a, preventing duplicates this way.

We set the color of a broker B to *black* when starting the reconfiguration until it has received a separator message that is originated from B's side of the reconfiguration cycle. Then, B changes its color to *gray*. A message adopts the color of the first broker on the reconfiguration cycle it passes. The color is removed again, when the message leaves the reconfiguration cycle. Thus, the separator message separates black messages from gray ones for the brokers on the reconfiguration cycle. A broker that receives a separator message from a neighbor knows that it will not receive a black message from this neighbor anymore for the rest of the reconfiguration. The color of separator messages is black, such that they are discarded when they arrive at a. Black messages are allowed to pass r when they arrive on R_1 or R_2 and are discarded silently otherwise. Similarly, only gray messages are routed over a. Since a message is either black or gray, it can only pass one of the two links r and a. Hence, it is guaranteed that there is always at most one forwarding path between any two brokers a message could follow and duplicates are prevented thus.

(Un)subscription Completeness. (Un)subscriptions that are issued during reconfiguration pass r or a according to their color. If one enters the reconfiguration cycle over a gray broker, it is forwarded normally but discarded at the broker on r due to its color. If a black (un)subscription reaches R_2 it is sent (e.g., out-of-band) to A_2, recolored to gray, and processed there as discussed above.

Ordering. Depending on the correctness requirements of the system, notification ordering must be maintained. Regarding control messages, it is always important to keep a FIFO-publisher ordering of related subscriptions and unsubscriptions such that, for example, a gray subscription cannot overtake an associated black unsubscription after the former has been sent from R_2 to A_2. Therefore, we introduce a queue $\mathcal{Q}_{sub}$ on A_2 that delays gray (un)subscriptions that were forwarded over a until the last black control message has been received from R_2. This is the case, when R_2 has received a separator message from its neighbor on R_1. To inform broker A_2 about that, R_2 signals A_2 the end of the reconfiguration. In the following, A_2 empties the queue and will not queue new gray messages anymore.

Notifications are queued similarly as control messages. Likewise, they are delayed on broker A_2, if forwarded over a, using a queue $\mathcal{Q}_{not}$ if they are gray. Messages from one publisher enter the reconfiguration cycle always over the same broker. Thus, two consecutive notifications from one publisher (N_1, for example) may be black and gray. While the black notification is routed over r, the gray notification has to be delayed on broker A_2 on the other side of a until it is guaranteed, that the black notification has reached every broker on A_2's side of the reconfiguration cycle. For *FIFO-publisher ordering*, gray messages have hence to be queued until a separator message is received from R_1. To maintain *causal ordering*, gray notifications have to be delayed until it is guaranteed that no black notification from both sides of the reconfiguration cycle can reach the broker anymore. From FIFO-ordering we know, that if A_2 receives a separator message from R_1, it is guaranteed, that no notifications from R_1's side are in transit on A_2's side of the reconfiguration cycle. To ensure, that there are no black messages in transit from A_2's side, the separator message forwarded by A_2 on the same side has to be copied and sent back over the reconfiguration cycle when it reaches the originating broker R_2 again. If A_2 has received both messages, this copy and R_1's separator message, it can start with processing the queue as it is guaranteed now, that there are no black messages heading towards A_2 on the cycle anymore.

2.3 Adding and Removing Leaf Brokers

Up to now, we discussed how to exchange a link with another. In this section, we present algorithms to add and remove leaf brokers.

2.3.1 Adding a Leaf Broker

A new leaf broker A has no subscriptions yet. Connecting one with a broker B that is already part of the network (as depicted in Fig. 2) thus means that B has to forward its subscriptions to A that are stored in its routing table and point to other neighbor brokers of B. Therefore, the regular routing algorithm is used. If simple routing is used, for example, all subscriptions in the routing table of B will be forwarded to A. For covering-based routing only those subscriptions are forwarded, that are not covered by others.

2.3.2 Removing a Leaf Broker

When a broker R is removed from the topology, like depicted in Fig. 2, its neighbor broker B simply has to remove every routing entry from its routing table that points to R and forward corresponding unsubscriptions if necessary according to the routing algorithm used.

2.4 Composite and Concurrent Reconfigurations

The types of reconfigurations defined in Sect. 2.1 are elementary. It is possible to carry out a sequence of elementary reconfigurations to realize one complex composite reconfiguration. For several reasons like optimizations, it might be useful or simply happens that reconfigurations are also carried out in parallel. We have to take care, that concurrent reconfigurations do not share any broker, as this can lead to serious problems regarding message ordering and completeness or even system correctness.

Therefore, we introduce a *lock-message* before starting the reconfiguration that has to be forwarded to every broker on the reconfiguration cycle. This guarantees that each broker can only take part in one reconfiguration at a time. Additionally the lock-message is used to initially color participating brokers black to prepare the reconfiguration process. If a lock-message of one reconfiguration encounters a broker that is already locked by another reconfiguration, it is sent back to the originating broker, freeing already locked brokers. To prevent livelocks caused by repeatedly colliding lock attempts additional measures must be applied (e.g., a randomized exponential backoff algorithm). After the reconfiguration is finished, an *unlock-message* releases all brokers on the reconfiguration cycle.

3 Evaluation

This section presents the results of a discrete event simulation[1] which we carried out to analyze the algorithm's behavior focusing on the difficult case of exchanging a link in the broker topology. The simulation was designed to both *verify* the correctness of the developed algorithm and to *evaluate* its performance in terms of reconfiguration overhead. The performance is measured in the number of control messages needed to complete a link exchange comparable to the strawman solution. Furthermore, we quantify the additional delay in the delivery of events, that is introduced by queueing notifications to guarantee their FIFO-publisher or causal ordering. We start with describing the simulation setup, before we discuss the results in detail.

3.1 Simulation Setup

The simulation is based on a transit-stub network topology with a total of 10,000 nodes, subdivided into 100 domains of equal size, that was generated using the BRITE [3] topology generator. The publish/subscribe infrastructure is then composed of 100 randomly chosen brokers which form an acyclic overlay network. The maximum delay of an overlay link is limited to 100 ms including the time needed by a broker to process a message. The brokers host a total of 500 clients, that are uniformly distributed among them. Each of the 50 publishers has 9 dedicated subscribers, that receive all of its notifications. The publication rate follows an exponential distribution such that a publisher is expected to create a new notification every 25 ms. The following experiments were repeated 25 times and arithmetic means as well as 95% confidence intervals were calculated for every measured value. To ensure, that results of different algorithms are comparable, the same setups and random seeds were used for corresponding simulations.

3.2 Measuring Reconfiguration Overhead

In the first experiment, we evaluate the control message complexity of the improved reconfiguration algorithm and compare it to the strawman approach. Therefore, 100 randomly chosen links are exchanged such that the network

[1]For the simulation source-code and setups refer to `http://kbs.cs.tu-berlin.de/~mjaeger/debs06/reconf.tar.gz`.

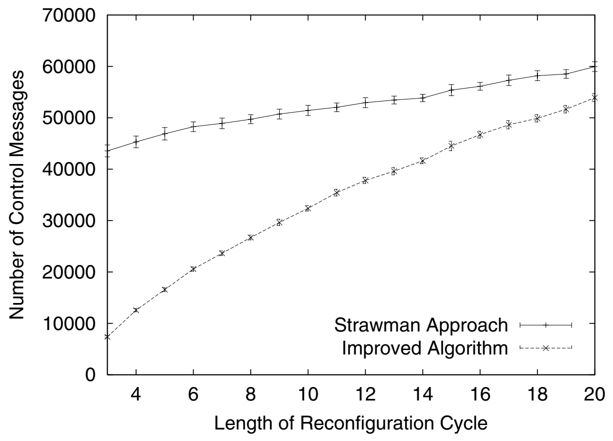

Figure 4. Number of control messages versus length of reconfiguration cycle.

stays connected and acyclic, while the control messages necessary to update the brokers' routing tables are counted. Furthermore, we check for the improved algorithm, whether any notification is lost or delivered twice to any of its subscribers to verify its correctness.

Figure 4 shows the number of control meassages as a function of the length of the reconfiguration cycle. The message complexity increases for both algorithms, because (un)subscriptions have to be forwarded to more brokers to update their routing tables, if the reconfiguration cycle grows. However, since the improved algorithm limits filter forwarding to the cycle, it clearly outperforms the strawman approch especially for small cycle lengths. Although the number of control messages increases, due to additional messages needed for coordination, the improved algorithm even remains more efficient for larger reconfigurations.

3.3 Measuring Ordering Delay

The improved reconfiguration algorithm ensures the complete delivery of notifications, but introduces a delay as notifications are queued after passing the new link, if their correct ordering has to be guaranteed. The delay depends on the type of ordering (FIFO-publisher or causal) and on the length of the reconfiguration cycle. Figure 5 shows the average delay of 100 random link exchanges measured as the mean time an affected notification is queued at the new link. The delay increases linearly with the length of the reconfiguration cycle, as seperator-messages have to travel over it before queued notifications are released again. Causal ordering is more expensive, because notifications are queued until separator messages have arrived from both sides of the cycle, while FIFO-publisher ordering only requires the receipt of one seperator-message from the opposite side.

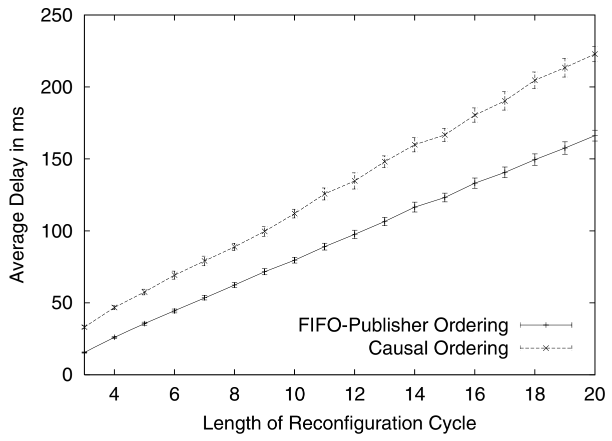

Figure 5. Average delay caused by ordering versus length of reconfiguration cycle.

4 Related Work

The work closest to ours is by Cugola et al. where they introduce the strawman approach discussed as the "uncoordinated approach" in [2]. The basic algorithm has later been improved by introducing a delay for issueing the unsubscriptions [2, 5]. With this solution, the new link is established a bounded delay (i.e., a timeout is used) before the old link is removed, i.e., subscription propagation starts earlier than unsubscription propagation. However, choosing a sensible value for the timeout seems difficult. To avoid the propagation of subscriptions that would otherwise be removed a short time later, subscriptions located at an endpoint of a removed link are removed from the routing tables of the respective brokers instantaneously and only their propagation is delayed.

In [1], the authors identify the *reconfiguration path* (a subset of the reconfiguration cycle) as the minimal portion of the system affected by the reconfiguration and reduce reconfiguration actions to this part. As a result, they are able to reduce the overhead dramatically and achieve subscription completeness. However, no guarantees are given for notification completeness, duplicates, and ordering. Concurrent reconfigurations are possible if the reconfiguration paths do not cross each other.

While related approaches assume that the old link is not available anymore, we still use it for message forwarding until the reconfiguration is finished. Thus, our solution addresses proactive scenarios, where reconfigurations serve administrative or optimizational purposes that should not disturb the normal operation of the publish/subscribe system. For these scenarios we also assume that the reconfiguration cycle is known in advance.

5 Conclusions

Reconfiguration in content-based publish/subscribe middleware is an important issue. In this paper, we investigated administrated reconfigurations that consist of elementary reconfiguration steps which include the exchange of a link as well as the addition or removal of a leaf broker. Algorithms were presented that prevent the loss or duplication of notifications and maintain the message ordering. For the ordering of notifications we focused on a FIFO-publisher and a causal ordering policy.

We carried out simulations and compared the number of control messages that are produced by the strawman approach with the overhead of our algorithm. The results show that our algorithm is more efficient in the scenarios tested. We also compared the average notification delay induced by FIFO-publisher and causal ordering.

Enabling reconfiguration in publish/subscribe systems is a prerequisite for supporting dynamic environments with adaptivity. Thus, the presented work is a building block for future adaptive publish/subscribe middleware that will lower administration effort and increase its applicability.

References

[1] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 1134–1140, Nicosia, Cyprus, 2004. ACM Press.

[2] G. Cugola, G. P. Picco, and A. L. Murphy. Towards dynamic reconfiguration of distributed publish-subscribe middleware. In *3rd International Workshop on Software Engineering and Middleware (SEM 2002)*, volume 2596 of *Lecture Notes in Computer Science*, pages 187–202, Orlando, FL, USA, 2002. Springer.

[3] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: An approach to universal topology generation. In *Proceedings of MASCOTS'01*, pages 346–353, Cincinnati, OH, USA, August 2001. IEEE Computer Society.

[4] H. Parzyjegla. Ein adaptives brokernetz für publish/subscribe systeme. Master's thesis, Berlin University of Technology, Berlin, Germany, October 2005. (in German), `http://www.kbs.cs.tu-berlin.de/publications/fulltext/diplParzyjegla.pdf`.

[5] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 03)*, pages 234–243, Providence, RI, USA, 2003. IEEE Press.