Minimizing the Reconfiguration Overhead in Content-Based Publish-Subscribe

Gianpaolo Cugola[1], Davide Frey[1], Amy L. Murphy[2], and Gian Pietro Picco[1]

[1] Dip. di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32, 20133 Milano, Italy
{cugola, frey, picco}@elet.polimi.it

[2] Dept. of Computer Science, Univ. of Rochester
P.O. Box 270226, Rochester NY, 14627, USA
murphy@cs.rochester.edu

ABSTRACT

Publish-subscribe middleware enables the components of a distributed application to subscribe for event notifications, and provides the infrastructure enabling their dynamic routing from sources to subscribers. This model decouples publishers from subscribers, and in principle makes it amenable to highly dynamic environments. Nevertheless, publish-subscribe systems exploiting a distributed event dispatcher are typically not able to rearrange dynamically their operations to adapt to changes impacting the topology of the dispatching infrastructure.

In this paper, we present a novel algorithm to deal with this kind of reconfiguration, which exploits a notion of *reconfiguration path* for minimizing the portion of the system affected by the reconfiguration. Simulations show that this approach significantly reduces the overhead involved with reconfiguration (up to 76%) with respect to a strawman solution available in the literature.

1. INTRODUCTION

In the last few years, publish-subscribe middleware has become popular, mostly because the asynchronous, implicit, multi-point, and peer-to-peer communication style it fosters is well-suited for modern distributed computing applications. While the majority of deployed systems is still centralized, commercial and academic efforts are currently focused on achieving better scalability by developing publish-subscribe middleware that exploits a distributed event dispatching infrastructure.

Beyond scalability, the next challenge for distributed publish-subscribe middleware is to introduce the ability to reconfigure the dispatching infrastructure to cope with changes in the topology of the physical network, and to do this without interrupting the normal system operation. The scenarios that involve a topological change are many. Some are determined by explicit changes in a controlled environment. For instance, this is the case in an enterprise environment whose system administrator decides to add new machines (and event dispatchers) to cope with an increased number of users. In other scenarios, however, the source of reconfiguration is not under control. This is the case in mobile computing scenarios, where wireless links enable changes in the topology of the physical network that may occur at any time as a result of mobility. In both cases, however, the publish-subscribe middleware deployed on top of the network being reconfigured must be able to automatically react to the topological changes.

The vast majority of currently available publish-subscribe middleware has ignored this reconfiguration problem, the only exceptions being a few systems that adopt an inefficient, strawman solution. We have previously tackled this problem in [14], with the goal of identifying a solution that could offer better performance with respect to the strawman approach and which could be applied to reconfiguration scenarios ranging from manually controlled to environmentally imposed. In this paper, we approach the same problem with a different goal: to push the optimization as far as possible, even if at the price of partially limiting applicability, e.g., by restricting it only to controllable reconfiguration scenarios.

The contributions of the paper can be summarized as follows. First, we define a notion of *reconfiguration path*, which identifies the minimal portion of the system involved in reconfiguration. Second, we present an algorithm that exploits this notion to reconfigure the system efficiently. Third, we present simulation results that compare the performance of the new algorithm against the strawman solution and our earlier solution [14]. The results show a significant overhead reduction, up to 76% if compared to the strawman solution.

The structure of the paper is as follows. Section 2 provides the reader with the basics of publish-subscribe middleware. Section 3 defines precisely the reconfiguration problem we tackle in this paper. Section 4 briefly describes the strawman solution mentioned above, introduces the notion of reconfiguration path, and describes the new algorithm. Section 5 presents the simulation results validating our algorithm, while Section 6 discusses its implications on the reconfiguration scenarios. Finally, Section 7 discusses related efforts in the field and Section 8 draws some conclusions and discusses future avenues of research.

2. PUBLISH-SUBSCRIBE SYSTEMS

Applications exploiting publish-subscribe middleware are organized as a collection of autonomous components, the *clients*, which interact by *publishing* events and by *subscribing* to the classes of events they are interested in. A component of the architecture, the *event dispatcher*, is responsible for collecting subscriptions and forwarding events to subscribers.

Recently, many publish-subscribe middleware have become available, which differ along several dimensions[1]. Two are usually considered fundamental: the expressiveness of the subscription language and the architecture of the event dispatcher.

The expressiveness of the subscription language draws a line between *subject-based* systems, where subscriptions identify only classes of events belonging to a given channel or subject, and *content-based* systems, where subscriptions contain expressions (called *event patterns*) that allow sophisticated matching on the event content. Our approach tackles a content-based subscription language, as this represents the most general and challenging case. Since content-based systems can be regarded as a generalization of subject-based ones, our approach is arguably applicable also to the latter.

The architecture of the event dispatcher can be either centralized or distributed; in this paper, we focus on the latter case. In such middleware, a set of *dispatchers* is interconnected in an overlay network and cooperatively route subscription and event messages, as in Figure 1. The systems exploiting a distributed dispatcher can be further classified according to the interconnection topology of dispatchers and the strategy exploited for message routing. In this work we consider a subscription forwarding scheme on an unrooted tree topology, as this choice covers the majority of existing systems.

In a subscription forwarding scheme [4], subscriptions are delivered to every dispatcher along a single unrooted tree spanning all dispatchers, and are used to establish the routes that are followed by published events. When a client issues a subscription, a message containing the corresponding event pattern is sent to the dispatcher the client is attached to. There, the event pattern is inserted in a subscription table together with the identifier of the subscriber, and the subscription is forwarded to all the neighbors. During this propagation, the dispatcher behaves as a subscriber with respect to the rest of the dispatching tree. Each dispatcher, in turn, records the event pattern and re-forwards the subscription to its neighbors, except for the one that sent it. This scheme is usually optimized by avoiding to forward multiple subscriptions for the same event pattern in the same direction[2]. This process effectively sets up a route for events, through the reverse path from the publisher to the subscriber. Requests to unsubscribe from a given event pattern are handled and propagated analogously to subscriptions, although at each hop entries in the subscription table are removed rather than inserted. This behavior is explained in more detail in Section 4, and a formalization is provided in the Appendix.

Figure 1 shows a dispatching tree where a dispatcher (the dark one) is subscribed to a certain event pattern. The arrows represent the routes laid down according to this subscription, and reflect the content of the subscription tables of each dispatcher. To avoid cluttering the figure, subscriptions are shown only for a single event pattern. To simplify the treatment, here and in the rest of the paper we ignore the presence of clients and focus on dispatchers. Accordingly, even if in principle only clients can be subscribers, with some stretch of terminology we say that a dispatcher is a subscriber if at least one of its clients is a subscriber. Moreover, we assume that the links connecting the dispatchers are FIFO and transport reliably subscriptions, unsubscriptions, events, and other control messages. Both assumptions are typical of mainstream publish-subscribe systems, and are easily satisfied by using TCP for communication between dispatchers.

[1]For more detailed comparisons see [4, 7, 15].

[2]Other optimizations are possible, e.g., by defining a notion of "coverage" among subscriptions, or by aggregating them, as in [4].

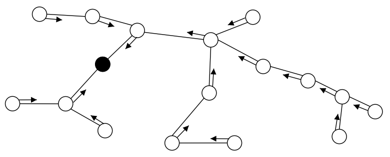


Figure 1: Subscription forwarding.

3. THE RECONFIGURATION PROBLEM

Publish-subscribe systems are intrinsically characterized by a high degree of reconfiguration, determined by their very operation. For instance, routes for events are continuously created and removed across the tree of dispatchers as clients subscribe and unsubscribe to and from events. Clearly, this is not the kind of reconfiguration we are investigating here. Instead, the dynamic reconfiguration we address can be defined informally as *the ability to rearrange the dispatching infrastructure to cope with changes in the topology of the underlying physical network, and to do this without interrupting the normal system operation.*

We view this problem as composed of three subproblems that involve:

1. the reconfiguration of the overlay network that realizes the dispatching infrastructure, to retain connectivity among dispatchers;
2. the reconfiguration of the subscription information held by each dispatcher, to bring it up-to-date with the changes above without interfering with the normal processing of subscriptions and unsubscriptions;
3. the minimization of event loss during reconfiguration.

The objective of the work we describe here, and of our earlier work described in [14], is to solve the second of the aforementioned problems. The rationale for this choice lies in the fact that *maintaining the consistency of subscription information is the defining problem of content-based publish-subscribe systems*: if the information necessary for event dispatching is misconfigured the whole purpose of a content-based system may be undermined. The availability of a tree connecting all the dispatchers is a precondition for the type of content-based systems we are interested in, but it is not their core feature. This, clearly, does not mean that we are disregarding the other two problems. Our ongoing activities are investigating how existing algorithms (e.g., those developed for MANET routing [16] or IP multicast) can be adapted to solve the first problem. Meanwhile we developed, on top of our solutions, a layer based on epidemic algorithms [6] and responsible for recovering events lost during reconfiguration, providing a solution to the third problem.

Within this general framework, in [14] we tackled the second problem without making any special assumption about either the sources of reconfiguration or the way the overlay network is kept connected. In this paper we adopt a rather different approach. Our goal here is (1) to push overhead optimization as far as possible and (2) to identify the requirements our solution places on the tree maintenance layer, and consequently the scenarios where it is immediately applicable. By pushing optimization to an extreme, we are interested in characterizing the whole spectrum of possibilities with respect to the problem of publish-subscribe reconfiguration. The promising results presented in the following sections demonstrate the effectiveness of our proposal, and therefore provide strong motivation to continue our investigation of how to exploit further the notion of reconfiguration path.

4. DEALING WITH RECONFIGURATION

As mentioned in the previous section, our goal is to find an algorithm to rearrange subscription information to keep it consistent with changes in the dispatching tree. This algorithm should minimize the network traffic involved in the reconfiguration.

To find such an algorithm, we observe that a tree of dispatchers may change in a number of ways. New dispatchers can be added, dispatchers can be removed, links may vanish or appear, and so on, in a potentially infinite number of possibilities. To identify a single solution that could be applied to every situation, we decided to focus on reconfigurations that involve the removal of a link and the insertion of a new one, thus keeping the dispatching tree connected. The reason for this choice is that link substitution represents the fundamental building block for more complex reconfigurations. For instance, the disappearance of a dispatcher from a tree could be easily dealt with as a number of link substitutions connecting the children of the dispatcher to its parent. At the same time, simpler reconfigurations, involving only link removal or insertion and thus leading to tree partitioning or merging, can be dealt with using plain subscriptions and unsubscriptions, as we describe later.

Given these premises, Section 4.1 describes a strawman approach to reconfiguration. Section 4.2 defines the notion of *reconfiguration path*, which identifies precisely the minimal portion of the tree that needs to be reconfigured. Finally, Section 4.3 presents our algorithm which exploits this notion to rearrange routes for events, thus reducing the traffic overhead involved in the reconfiguration.

4.1 A Strawman Approach

In principle, the removal of an existing link or the insertion of a new one can be treated by using exclusively the primitives already available in a publish-subscribe system, leveraging off of the high degree of decoupling among the dispatchers in the tree. In particular, the removal of a link can be dealt with by using unsubscriptions. When a link is removed, each of its end-points is no longer able to route events matching subscriptions issued by dispatchers to the other side of the tree. Hence, each of the end-points should behave as if it had received, from the other end-point, an unsubscription for each of the event patterns the latter was subscribed to. Similarly, the insertion of a new link in the tree can be carried out in a dual way using subscriptions. This approach is the most natural and convenient when reconfiguration involves only either the insertion or the removal of a link, and is actually adopted by some publish-

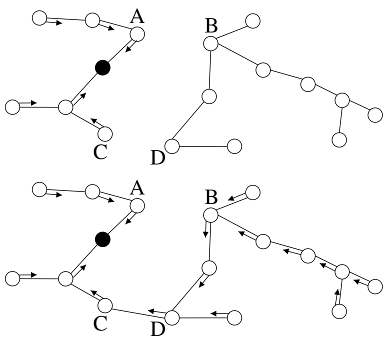


Figure 2: The dispatching tree of Figure 1 during and after a reconfiguration performed using the strawman approach.

subscribe middleware to deal with merging or partitioning.

Nevertheless, as mentioned earlier, in many other situations it is desirable to replace a link with a new one, thus effectively reconfiguring the topology of the tree while keeping the same nodes as members of the tree. This case can be dealt with by some combination of the add and remove link operations, e.g., as suggested in [4, 23], but the results are far from optimal. In fact, if the route reconfigurations caused by link removal and insertion are allowed to propagate concurrently, they may lead to the dissemination of subscriptions which are removed shortly after, or to the removal of subscriptions that are subsequently restored, thus wasting a lot of messages and potentially causing far reaching and long lasting disruption of communication.

Figure 2 illustrates this concept on the dispatching tree of Figure 1. For simplicity, the figure shows subscriptions for a single event pattern p, but in a real system the reconfiguration algorithm would process all event patterns in parallel. The Appendix provides a formalization of the subscription and event processing operations for a publish-subscribe system that adopts a subscription forwarding scheme and of the operations to add and remove a link between dispatchers.

According to this strawman solution, when the link between A and B is removed, both end-points trigger unsubscriptions in their subtrees, without taking into account the fact that a new link has been found between C and D. Depending on the speed of the route destruction and construction processes, subscriptions in B's subtree may be completely eliminated, since there are no subscribers for p in that tree. Nevertheless, shortly afterwards most of these subscriptions will be rebuilt by the reconfiguration process. A similar problem arises if subscriptions propagate before unsubscriptions.

The drawbacks of this approach are essentially caused by a single problem: the propagation of reconfiguration messages reaches areas of the dispatching tree that are far from the ones directly involved in the topology change, and which should not be affected at all. This observation leads to the idea of delimiting the area involved in the reconfiguration, a key element of our approach.

4.2 Delimiting the Reconfiguration

When an existing link must be replaced with a new one, we can identify specifically the portion of the dispatching

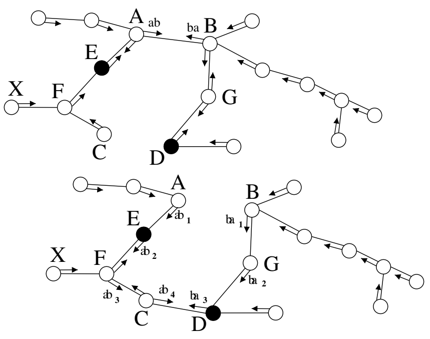


Figure 3: A dispatching tree before and after a reconfiguration.

tree affected by the reconfiguration. This *reconfiguration path* can be defined as the concatenation of three sequences of dispatchers:

- the *head path* is the sequence that starts with the first end-point of the removed link, and contains all the dispatchers connecting it to the end-point of the new link that lies in the same subtree. The head path is empty if the first end-point of the removed link and the end-point of the inserted link coincide;
- the *new link* is an ordered pair constituted by the end-points of the link being inserted in the tree;
- the *tail path* is the sequence that ends with the second end-point of the removed link, and contains all the dispatchers connecting it to the end-point of the new link that lies in the same subtree. The tail path is empty if the second end-point of the removed link and the end-point of the inserted link coincide.

The above definition requires the ability to establish an ordering between the end-points of the vanished link, e.g. exploiting mechanisms specific to the environment such as IP addresses. Here it suffices to observe that such an ordering, combined with the topological properties of the tree, makes the reconfiguration path a directed path.

Figure 3 shows an example of reconfiguration where the link (A, B) is being substituted with the link (C, D). In this case, the head path is (A, E, F), the new link is (C, D), the tail path is (G, B), yielding (A, E, F, C, D, G, B) as the reconfiguration path.

The decoupling between dispatchers, combined with the notion of reconfiguration path, are the keys to limit the scope of the reconfiguration process. In fact, each dispatcher routes events and subscriptions only based on the local knowledge gathered from its neighbors; similarly, its actions are limited to messages sent only to its immediate neighbors. In other words, each dispatcher has knowledge only about its immediate "next hops". For instance, there is no way for X to know that a given event matching a pattern p is ultimately destined to E and D. All X knows is that F subscribed to receive events matching p. The distributed knowledge throughout the dispatching tree steers the event towards its destinations, E and D in this case.

For this reason, *a dispatcher that does not belong to the reconfiguration path will not experience a change in its subscription tables.* It will continue forwarding events the same way it was doing before, i.e., towards the next hop, according to the information in its subscription table. This information needs to be changed only on the dispatchers lying on the reconfiguration path. In Figure 3, for instance, X will keep forwarding events to F as before the reconfiguration.

4.3 Performing the Reconfiguration

To reconfigure the tree, events that used to be routed through the removed link must now be routed through the new link and hence along the reconfiguration path. Thus, subscriptions that were exploiting the vanished link must now be replaced by subscriptions along the reconfiguration path. In Figure 3, the subscription ab, that was exploiting the vanished link (A, B) to route events to D, is removed by the reconfiguration; the routing it provided is now performed equivalently by subscriptions ab_1, ab_2, ab_3, and ab_4. Similarly, the effect formerly achieved by subscription ba is now obtained by ba_1, ba_2, and ba_3 that, together with the subscriptions already present in between C and E, enable events to reach the subscriber E. Note how, in the particular tree configuration shown in the figure, only ab_2, ab_3, and ab_4 need to be added towards B; ab_1, instead, was already present to route events from A towards the subscriber E. Analogously, only ba_3 needs to be added towards A, since the other subscriptions were already present because of D.

From Figure 3 it can be seen how the subscriptions that replace ab are needed only on the head path and on the first end-point of the new link, i.e., from A to C. In fact, these subscriptions are needed only to route events originating in A's subtree into B's subtree. In the latter subtree, subscriptions are already in place to further propagate these events to the proper subscribers, unless an unsubscription occurred during disconnection. Similar reasoning holds for the tail path, i.e., from B to D.

This observation can be used to determine how to propagate subscriptions along the reconfiguration path. The reconfiguration process we adopt propagates along dispatchers only in one direction, following the ordering imposed by the reconfiguration path. Nevertheless, it must correctly reconstruct subscriptions that are headed in both directions, from A to B and vice versa. Thus, the process must be somehow different in the head and tail paths, and on the new link. To complicate the process further, the normal operations of the publish-subscribe system are also being carried out during the reconfiguration. Still, the reconfiguration must leave the system in a consistent state.

In the remainder of this section we present a reconfiguration process that leverages off of the definitions and observations made thus far. For the sake of clarity, the description of the algorithm is split in two parts. Section 4.3.1 describes the basic operations of the algorithm without the details concerning the management of the subscriptions and unsubscriptions issued during the reconfiguration; these are instead presented in Section 4.3.2. A formal description of the complete algorithm is provided in the Appendix.

4.3.1 Basic Operation of the Algorithm

This section steps through a single reconfiguration from initialization, through the head path, across the new link, and finally on the tail path. Note that in describing the algorithm we assume that we are dealing with one reconfiguration at a time. Multiple, concurrent reconfigurations can be dealt with by propagating a reconfiguration identifier in the control messages, as shown in the formalization in the Appendix. We elaborate further on this issue in Section 6.

Starting the Reconfiguration. The reconfiguration process is started by the first of the end-points of the link being removed from the dispatching tree. It is also the first dispatcher on the reconfiguration path, referred to as the *initiator* of the reconfiguration.

The initiator starts the reconfiguration by determining the event patterns belonging to subscriptions previously issued by the other end-point of the vanished link (ab in Figure 3). Two sets of patterns are relevant: the set P_{add} of patterns for which subscriptions need to be added along the reconfiguration path, and the set P_{del} of patterns for which subscriptions need to be removed along the reconfiguration path since they were formerly used *only* to route events across the removed link. Looking at Figure 3, essentially P_{add} enables the insertion of the ab_i subscriptions that are missing on the path from A to C, while P_{del} enables the removal of the unnecessary subscriptions on the path from C to A.

At the initiator, these two sets are coincident and are used to modify the subscription table accordingly. For each event pattern in P_{add}, a new entry is inserted in the table as if it were a subscription coming from the next dispatcher in the reconfiguration path (E in Figure 3). All the entries in P_{del} that were associated with a subscription towards the other end-point of the removed link (that would originally direct events towards B in Figure 3) are deleted from the subscription table. In Figure 3, this step causes the deletion of ab and the insertion of ab_1 in the subscription table of A. In the formalization in the appendix, this processing occurs when the initiator emulates a reconfiguration message to itself from the other end-point of the vanished link.

Reconfiguring the Head Path. At this point, the reconfiguration of the initiator's subscription table is complete, and processing can proceed along the reconfiguration path. The next step involves computing a new set P_{del}. In general, P_{del} is a subset of P_{add}, since it must contain the subscriptions that are used *only* to route the events towards the removed link; subscriptions that happen to be on the reconfiguration path but are also needed to route events towards different areas of A's subtree should not be removed. This information is computed at each dispatcher by looking at its subscription table: patterns in P_{del} for which a subscription exists towards a dispatcher or a client other than the next one on the reconfiguration path are removed from P_{del}, shrinking it during its travel through the head path.

The two sets of patterns P_{add} and P_{del}, together with the reconfiguration path, are placed in a reconfiguration message RECMSG. This message originates at the initiator, and is propagated along the reconfiguration path, starting at the initiator. Each dispatcher, upon receiving RECMSG, performs the same operations originally performed by the initiator: first, the subscription table is updated, then P_{del} is recomputed, and finally a new RECMSG is propagated.

Reconciling Subscriptions Across the New Link. When the reconfiguration process reaches the end-point of the new link that belongs to the initiator's subtree (C in Figure 3), the new link is already physically available but it has not been logically "activated" in the tree by the reconfiguration process.

Therefore C updates its subscription table as described earlier, activates the new link, and sends to the second end-point a subscription message for each event pattern in its subscription table, followed by the RECMSG containing P_{add}.

Completing the Reconfiguration. At this point, the subscriptions sent across the new link propagate throughout the second subtree as normal subscriptions. Most importantly, they enable the correct routing of events across the new link and establish the path for event propagation on the tail path.

The only remaining step is the removal of superfluous subscriptions on the tail path, i.e., those subscriptions that were used only to route events to the first subtree via the removed link (e.g., the subscription from G to B and the one from D to G in Figure 3). These subscriptions cannot be removed until the subscription messages have propagated all the way to the end of the reconfiguration path (B in Figure 3), since they could also be needed by some other subscribers in the second subtree, and this information is not known until this condition has been checked by all the dispatchers on the tail path. Hence, the second end-point of the new link propagates a RECMSG along the tail path. When this RECMSG is finally received by the last dispatcher on the reconfiguration path, the latter behaves as if it had received a request to remove the link to the initiator, hence generating an unsubscription message for each subscription in its table issued by the initiator. This eliminates superfluous subscriptions from the reconfiguration path, without the risk of removing necessary ones, because the construction part of the reconfiguration process has already been completed.

4.3.2 Dealing with Concurrent (Un)Subscriptions

This section completes the description of the algorithm by appropriately tackling some issues arising when subscriptions and unsubscriptions are issued concurrently with the reconfiguration.

Avoiding Race Conditions on the Head Path. The first case in which the normal behavior of the system can interfere with the reconfiguration process is on the head path. Along the head path, in fact, RECMSG propagates in the opposite direction with respect to the established subscriptions, contrary to the usual way of processing subscriptions.

Let us refer to the sender of RECMSG as S and to its recipient as R. Normally, a subscription message subscribes the sender to events sent by the recipient. The subscription becomes active when the recipient processes the message and inserts a proper entry into its subscription table. Instead, the recipient of RECMSG becomes subscribed to events sent by the sender, which has already inserted the recipient in its subscription table. In other words, R finds itself subscribed to certain events, and does not know this fact until it receives RECMSG from S.

This can cause problems if R processes an unsubscription while RECMSG is being sent towards it. In fact, it could decide to unsubscribe from S, before knowing that it should instead keep the subscription as a result of the reconfiguration process. It is important to note that an unsubscription causing this problem can be originated only by a subscriber in the subtree of the head path, since, at this point connectivity to the other subtree has not yet been resumed.

A viable solution is to have S ignore unsubscription messages coming from R, if they refer to a pattern needed by the reconfiguration process. This solution can be implemented by means of an *ignore table*, I, containing the event patterns whose unsubscriptions should be ignored. This way, S can check the ignore table before attempting to process an unsubscription coming from R. Moreover, an additional control message, CTRLMSG, is needed to allow a dispatcher on the head path to notify its upstream neighbor that RECMSG has been processed, and the normal processing of unsubscriptions can be restored.

The unsubscriptions that need to be ignored are those related to event patterns for which subscriptions are being added by the reconfiguration, and are already present due to a previous subscription. Thus, S should include these patterns in its ignore table before forwarding RECMSG. However, it is not enough to consider only the subscriptions present when RECMSG is processed. In fact, S could receive a subscription message from R, immediately followed by an unsubscription, before RECMSG reaches R. In this case the subscription would cause no action, since it would already have been added by the reconfiguration, but the unsubscription would instead be processed, thus stealing the subscription added by the reconfiguration.

A conservative solution is therefore to add to the ignore table all the patterns in P_{add} before forwarding RECMSG, regardless of whether its recipient is already subscribed to them, and to remove them from the ignore table when the control message is received. In this case the ignore table grows larger than needed, but no special processing is required when a subscription is received.

Reconciling Subscriptions Across the New Link. Another problem caused by subscriptions and unsubscriptions issued during the reconfiguration arises as soon as RECMSG arrives at the new link and the two subtrees are effectively joined. In fact, up to that point, the reconfiguration was confined only to the initiator's subtree, while the operations in the other subtree continued normally. In particular, subscriptions and unsubscriptions could propagate through the second subtree without the possibility to reach the initiator's subtree. The views in the two subtrees must therefore be reconciled.

Let us begin by considering unsubscriptions which may have been issued in the second subtree during the first part of the reconfiguration. When the second end-point of the new link (D in Figure 3) receives RECMSG, it can determine which subscriptions have been laid out in the first subtree that are now superfluous, since they were needed only to route events towards subscribers in the other portion of the tree that unsubscribed meanwhile. This set of event patterns is obtained by removing all the patterns that are not in the second end-point's subscription table from the P_{add} set contained in RECMSG. For each of these event patterns, the second end-point sends an unsubscription to the first subtree.

The only other step is the propagation to the first subtree of the subscriptions issued in the second subtree during the first part of the reconfiguration. Again, this can be done by comparing the event patterns in P_{add} against the second end-point's subscription table. Subscriptions for event patterns which are in the subscription table but not in P_{add} can then be sent back to the first subtree.

Propagating these subscriptions to the first subtree as soon as RECMSG reaches the second end-point of the new link leads to a correct layout, but may cause some subscriptions to be added and then removed at the end of the reconfiguration. Some of the subscriptions contained in the second end-point's subscription table, in fact, may not be needed anymore because they were only used to route events to some subscriber in the first subtree via the old link. Propagating these subscriptions to the first subtree would therefore be unnecessary and would cause them to be removed by the unsubscriptions issued as the last step of the reconfiguration.

To avoid the unnecessary propagation of these subscriptions to the first subtree, it is necessary to wait until the unsubscriptions propagated as the last step of the reconfiguration reach the second end-point of the new link.

Therefore, when the second end-point of the new link receives RECMSG, it will still send unsubscriptions to the first subtree for the patterns in P_{add} which do not have a match in its subscription table; in addition, it will store P_{add}, propagate RECMSG, and wait until it is notified, as described in the following, that the reconfiguration process has completed.

The reconfiguration message is propagated along the tail path as described in Section 4.3.1. When the second endpoint of the old link receives it, it first starts the unsubscription phase by sending unsubscriptions along the tail path as described earlier. Subsequently, it sends a special FLUSHMSG along the tail path to notify the second end-point of the new link that the reconfiguration process has been completed. When the second end-point of the new link receives this FLUSHMSG, it has already processed the unsubscriptions propagated as the last step of the reconfiguration. Therefore it retrieves the P_{add} stored previously and compares it with its own subscription table. If there are any patterns not in P_{add} for which its subscription table contains subscriptions not issued by the other end-point of the new link, it propagates those subscription across the new link to the first subtree, completing the reconfiguration process.

5. SIMULATION RESULTS

This section presents an evaluation of our algorithm by means of simulations which compare it to two other approaches. The baseline for our comparisons is the strawman solution described in Section 4, whereas the other algorithm we consider is the one presented in [14]. This latter algorithm enhances the strawman solution in a number of ways, in particular by introducing mechanisms to perform unsubscriptions only after subscriptions, thus reducing the disruption of routes. For this reason, hereafter we refer to this approach with the oxymoron *"optimized strawman"* solution. The reader can find the complete description of this algorithm, together with a thorough evaluation through simulation, in [14].

The simulations we performed were designed with two different goals in mind. The first was to *verify* that our algorithm behaves correctly in the presence of reconfiguration. The second was to *evaluate* the performance of our algorithm in terms of minimizing overhead with respect to the two previously described solutions. These two goals were satisfied by two sets of simulations, measuring the percentage of events correctly delivered, and the number of messages exchanged during the reconfigurations.

5.1 Simulation Setting

Published work on publish-subscribe systems rarely presents validation of results through simulation. In the absence of reference scenarios for comparing these systems we defined our own, based on what we believe are reasonable assumptions covering a wide spectrum of applications. The simulation setting is the same we used in [14], which allows us to easily compare the results.

Events, subscriptions, and matching. Events are represented as randomly-generated 9-character strings, where each character can be any of the (96) printable characters. Subscriptions are represented as a single character. An event matches a subscription if it contains the character specified by the subscription. Hence, 96 different subscriptions are available in the system. Nevertheless, each dispatcher can subscribe to at most s subscriptions drawn randomly from the 96 available. In our simulations, we chose s=10, since in a content-based system it is unlikely that a subscription is shared among a large number of subscribers.

Publish rate. The behavior of each dispatcher in terms of publish and (un)subscriptions is governed by a triple of parameters, f_{pub}, f_{sub}, and f_{unsub}, respectively governing the frequency at which publish, subscribe, and unsubscribe operations are invoked by each dispatcher. The most relevant is f_{pub}, which essentially determines the system load in terms of event messages that need to be routed. Based on this parameter, we defined two load scenarios: one with a relatively low publish rate ($f_{pub} = 0.001$, about 1 publish/s), and one with a large number of events ($f_{pub} = 0.05$, about 50 publish/s). To put these values in context, the publish rate of applications dominated by human interaction, such as collaborative work in mobile environments or publishing files in a peer-to-peer network, should be comparable to, if not lower than, 1 publish/s.

Density of subscribers. The extent to which (un)subscriptions are propagated is determined by the density of subscribers in the tree: the more subscribers, the less a subscription travels. We defined two scenarios: one with sparse subscribers (20% of the dispatching tree), and one with dense subscribers (80% of the tree).

Tree topology. The results we present here are all obtained with tree configurations up to 200 dispatchers, each connected with at most four others. Clients are not modeled explicitly, as their activity affects only the dispatcher they are attached to.

Tree reconfiguration. In our simulations we are not interested in cases where a dispatching tree becomes partitioned or merged with another, as this case is handled in the same way by all three solutions. Instead, we evaluate the performance of the algorithms when a broken link is eventually replaced by another. The selection of the links breaking or appearing is done randomly. However, the same random sequence was applied to all the three algorithms. To retain some degree of control about when a reconfiguration occurs, we assume that each broken link is replaced by a new one in $0.1s$. While this is not necessarily a good representation of reality, the bias introduced by this assumption affects the three approaches in roughly the same way, and hence does not influence our results.

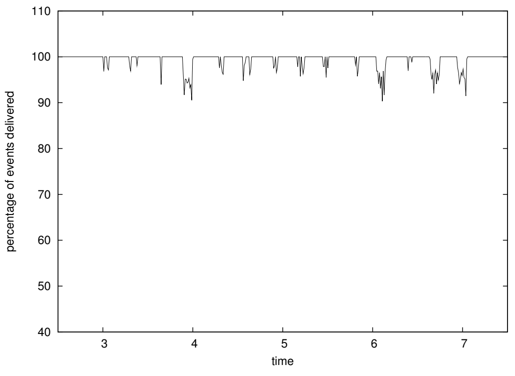


Figure 4: Event delivery during reconfigurations.

Reconfigurations are allowed only in the interval between 3 and 7 seconds, with a frequency determined by the duration of the interval between two reconfigurations, ρ. In the scenarios we consider, this interval is set to the value $\rho = 0.3s$, which yields non-overlapping reconfigurations.

Reducing the effect of randomization. Since topology, subscriptions, events, and reconfigurations are determined randomly, our results had a significant degree of variability. To reduce the bias induced by randomization, we ran each configuration 30 times using different seeds, and then averaged the results. Nevertheless, for each configuration the same seed was used for comparing the three approaches.

Simulation tool. In this paper we compare the algorithms only at the application level, and we are not concerned about the underlying networking stack. Our simulations use OMNET++ [21], a free, open source discrete event simulation tool.

5.2 Measuring Event Delivery

Before we proceed, it is important to note that we do not regard event delivery as a performance metric for our solution. As we discussed earlier, in this paper we focus only on identifying more efficient ways to restore the subscription information necessary to route events and not to prevent event loss, although this latter topic is the subject of our ongoing research [6]. Instead, we measure event delivery to verify that our algorithm is capable of restoring a correct, loop-free routing of events after each reconfiguration. If the algorithm behaves correctly, the percentage of events delivered should drop temporarily as a consequence of reconfiguration, and then come back to exactly 100%. This is indeed the behavior resulting from our simulations, as shown in Figure 4.

The measurements were performed by relying on a subset of the dispatchers belonging to a *stable core*. Dispatchers in the stable core have a more constrained behavior than the others, since they are forbidden to (un)subscribe after a given time threshold σ, whose value is set to $2s$ in our tests. Measurements about event delivery are constructed by computing the ideal set of recipients for each published message, and comparing it with the actual number of copies received.

The ideal set of recipients can be computed easily based on knowledge of the subscription tables of the dispatchers in the stable core, since this information is required to remain stable after σ. Moreover, since only the stable core is subject to this limitation, the algorithm is validated not only against the reconfiguration coming from changes in the topology, but also for the reconfiguration of routing information determined by the (un)subscriptions coming from dispatchers not in the core. The results in Figure 4 are derived with 100 dispatchers, 50% of which belong to the core. Moreover, 50% of the dispatchers inside the core, and 50% of those outside the core are subscribers. The event load is assumed to be large, with $f_{pub} = 0.05$ (i.e., about 50 publish/s).

In this chart, we do not report the measurements for the other two algorithms, since the simulations showed that the behavior of the three approaches is essentially the same.

5.3 Measuring Reconfiguration Overhead

The amount of reconfiguration overhead is the metric that we need to compare our solution against the other two.

The overhead is determined by the sum of three components: (1) the (un)subscription messages being exchanged because of reconfiguration; (2) the event messages being misrouted along obsolete subscriptions; and (3) the additional messages required by our solution in order to limit the changes to the reconfiguration path. The first two components are present in all the solutions we consider, whereas the third is present only in our approach.

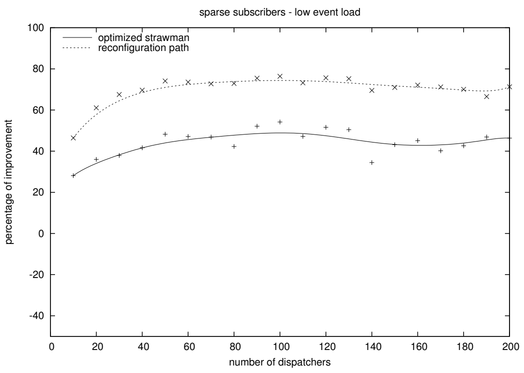
The overhead generated by messages is calculated in terms of the number of hops they have traveled. Thus, for instance, a subscription issued by a dispatcher generates an overhead equal to the number of hops traveled by the subscription message. Moreover, overhead is measured by considering every dispatcher in the network as part of the stable core defined previously for the measurement of event delivery. This way, the only (un)subscription messages exchanged in the system are those caused by reconfiguration. Note how this is actually a conservative situation: in fact, we verified the intuition that the percentage of improvement is actually larger when there are additional (un)subscriptions being injected in the system concurrently to reconfigurations.

Since the messages contributing to the overhead differ from each other, they were assigned a weight based on an estimate of the (traffic) cost incurred during forwarding from one dispatcher to the others. Events were considered as the base for our cost model and were assigned a weight of 1.0. Subscriptions and unsubscriptions were also assigned a cost equal to 1.0. Each RECMSG, on the other hand, contains two arbitrarily large sets of event patterns: therefore its weight was evaluated on the basis of the sizes of these sets. Given a RECMSG containing the two sets of patterns P_{add} and P_{del}, its weight was computed as:

$$weight(\textsc{recmsg}) = weight(\textsc{sub}) * (\frac{2(|P_{add}| + |P_{del}|)}{3} + \frac{1}{3})$$

where $|X|$ is the cardinality of set X and $weight(\textsc{sub})$ is the weight of a subscription containing one pattern. This weight can be explained by observing that the cost associated to sending a RECMSG can be divided into a fixed costs determined by the size of the message header and by the information associated to the reconfiguration path[3], and a variable cost associated to the event patterns contained in P_{add} and P_{del}. The sum of these costs should be comparable to that of an (un)subscription if the message contains a single event pattern, whereas it should be less than the one of sending an (un)subscription for each of the event patterns in the two sets, when several patterns are present. The CTRLMSG is a simple acknowledgment sent to the preceding dispatcher along the reconfiguration path, and contains only the reconfiguration identifier. Its weight was therefore set to 0.1. Finally, the FLUSHMSG does not contain any event patterns either, although it is slightly larger than the CTRLMSG because it needs to contain the addresses of the dispatchers in the reconfiguration path. Hence, its weight was set to 0.2

[3]The cost associated to sending the reconfiguration path was approximated with a constant.



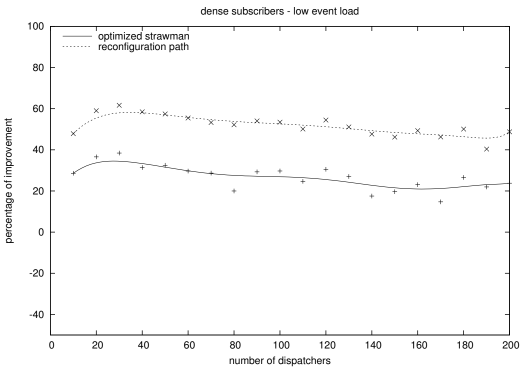


Figure 5: Percentage of improvement with respect to the strawman solution with a low (top) and a high (bottom) density of subscribers.

Overall improvement. Figure 5 shows the percentage of improvement of both our solution and the optimized strawman with respect to the strawman solution, in a configuration with $f_{pub} = 0.001$. As in [14], the original data points are reported together with their Bezier interpolation to help visualize the overall trend.

By showing the results for dense (80%) and sparse (20%) configurations of subscribers, the charts in Figure 5 confirm the intuition that both the optimized strawman and our algorithm perform better in scenarios with a low density of subscribers. In this case, the unnecessary (un)subscriptions propagated by the strawman solution travel farther in the network than with the other two algorithms. In addition, the percentage improvement of our approach with respect to the optimized strawman is greater for sparse rather than

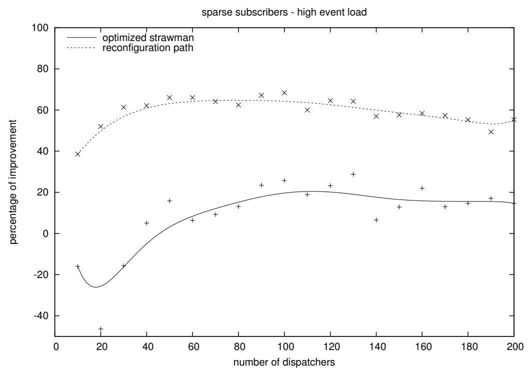


Figure 6: Percentage of improvement with respect to the strawman solution with a large number of events and a low density of subscribers.

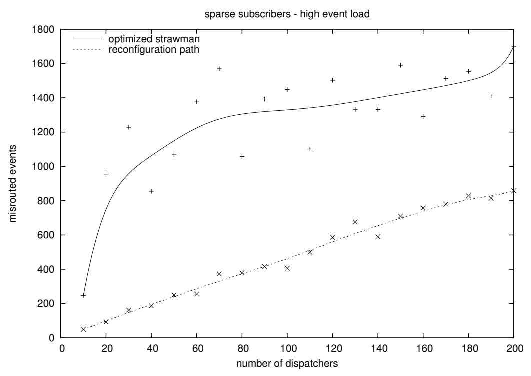


Figure 7: Misrouted events in a scenario with sparse subscribers and a high event load.

dense trees. This is because our approach limits the reconfiguration strictly to the reconfiguration path, while the optimized strawman algorithm still causes some subscriptions to travel to areas of the network from which they are later removed.

To give a feel of the relevance of the percentage improvement, in a dense tree with 200 dispatchers there are on average 3,500 messages (with a peak of 8,000) being exchanged during the reconfigurations using the strawman approach, while in a sparse tree there are on average 5,300 (with a peak of 18,000). Hence, even the smallest improvement provided by the optimized strawman—15% in a dense tree—already leads to a significant reduction in the traffic overhead. The improvement of 40% provided by our solution in the same setting reduces the cost of reconfiguration even further.

Figure 6 shows how the overhead is affected by a low density of subscribers when the event load is high. Comparing this to the top of Figure 5, the percentage improvement of both our solution and the optimized strawman solution is less than when the event load is low. The reason for this is that a high publish rate increases the overhead caused by misrouted events, a phenomenon we analyze next.

Misrouted Events. When events are published at a high rate, the overhead becomes more dependent on the number of misrouted events, because more events can be forwarded along stale routes not yet removed by the reconfiguration. The solution which suffers most from this phenomenon is the optimized strawman, since in this approach unsubscriptions are deferred, and hence obsolete routes are maintained for a longer period of time. Instead, our solution exhibits a lower overhead than the optimized strawman because the stale subscriptions which can cause the propagation of misrouted events are situated only along the reconfiguration path. Figure 7 shows graphically the difference in the number of misrouted events generated by our solution and by the optimized strawman algorithm, in a scenario with high event load ($f_{pub} = 0.05$).

Going back to Figure 6 and the top chart in Figure 5, we can now appreciate better the impact of misrouted events. The improvement obtained by the optimized strawman drops from an average of about 44% with few events to an average of about 10% with a large number of events. Instead, the improvement achieved by our solution only decreases from 70% to 60%.

6. DISCUSSION

The results presented in the previous section demonstrate that our initial goal of pushing optimization to an extreme has been achieved. The identification of the reconfiguration path as the minimal portion of the tree of dispatchers to be involved in the reconfiguration guided us in defining an algorithm that performs very well even with respect to the optimized solution described in [14]. On the other hand, the peculiarities of this solution pose some requirements on the tree maintenance algorithm which, in turn, suggest specific scenarios of reconfiguration.

In particular, the new algorithm presented here requires a tree maintenance algorithm capable of detecting link breakage *and* of determining a new route that could replace the broken link (i.e., the reconfiguration path). Moreover, in Section 4 we mentioned that we can take into account concurrent reconfigurations by distinguishing them through a reconfiguration identifier, as shown in the formalization in the Appendix. However, we implicitly assumed that the resulting reconfiguration paths do not overlap. This is indeed a limitation of our algorithm: the processing of concurrent reconfigurations would interfere if performed along overlapping reconfiguration paths.

These considerations suggest that the scenarios where our algorithm finds immediate and practical applicability are those involving a reconfiguration triggered at the application layer, e.g., when reconfiguration is initiated by a system administrator for balancing the traffic load or to add new dispatchers. In these controlled environments the above requirements are easily met, and our algorithm provides the best performance in terms of traffic overhead due to reconfiguration. The applicability of our approach is strengthened further by the observation that these controlled scenarios are very common in the domains where publish-subscribe middleware are currently deployed, i.e., large enterprise networks.

Another scenario that has recently been attracting the attention of both developers and researchers and that can exploit our approach is provided by peer-to-peer applications [13]. Well-known problems in this field include how to route information among peers (e.g., queries and replies in file sharing applications, or messages in messaging applications) and how to reconfigure routes to cope with the frequent changes of the topology of the peer overlay network as users join and leave the system. Here is where our approach can be applied. In fact, as observed also by other researchers [9], content-based routing may be easily adapted to this setting by providing each peer with the ability to route messages much like a dispatcher. This scenario perfectly fits the constraints identified above since connection and disconnection of peers usually is kept under strict control of the application, e.g., users must press a button to join or leave the peer-to-peer network.

More problems exist in less controlled scenarios, such as those resulting from the adoption of mobile, wireless networks. In these settings, reconfiguration is largely out of control of the application and it is difficult to avoid concurrent, overlapping reconfigurations. Currently, our approach does not solve these issues, but we are investigating alternative mechanisms exploiting to some extent the notion of reconfiguration path identified for the first time in this paper. On the other hand, it is worth noting that the solution we presented in [14] does fit these more extreme situations. Together, the two approaches cover the whole spectrum of reconfiguration: the solution presented here brings maximum performance in controlled scenarios, while the one presented in [14] still brings significant improvements but for arbitrary reconfiguration. Clearly, the choice of whether to use one or the other depends on considerations tied to the deployment environment.

7. RELATED WORK

Most publish-subscribe middleware are targeted to local area networks and adopt a centralized dispatcher. In recent years, the problem of wide-area event notification attracted the attention of researchers [19] and systems exploiting a distributed dispatcher became available, such as TIBCO's TIB/Rendezvous, Jedi [7], Siena [4], READY [8], Keryx [22], Gryphon [1], and Elvin4 [17]. To the best of our knowledge, none of them provides any special mechanism to support the reconfiguration addressed by this paper.

The closest match is the work on Siena [4] and the system described in [23]. These papers briefly suggest the use of the strawman solution to allow subtrees of dispatchers to be merged or trees to be split, but they do not provide details about its design, let apart providing an implementation or a validation through simulation. Jedi [7] provides a different form of reconfiguration that allows only clients (not dispatchers) to be added, removed, or moved to a different dispatcher at run-time. Similarly, Elvin supports mobile clients through a proxy server [18]. Finally, some research projects, such as IBM Gryphon [1] and Microsoft Herald [3], claim to support a notion of reconfiguration similar to the one we address in this work, but we were unable to find any public documentation about existing results.

In addition to the area of publish-subscribe middleware, IP multicast routing protocols and group communication middleware are also related to the topic of this paper.

The purpose of IP multicast is to provide efficient datagram communication services for applications that need to send the same data to a group of recipients. Typical examples are audio and video streaming. This goal results in routing strategies that largely differ from the one adopted in publish-subscribe systems. In particular, applications based on multicast exploit a finite number of statically known groups, while in content-based publish-subscribe systems the "groups" (i.e., the event patterns) are potentially infinite and not statically known. Moreover, IP multicast groups are disjoint and each packet is explicitly addressed to a single group, while in the system we are interested in addressing is based on message content, and hence a message can match multiple subscriptions. Finally, publish-subscribe usually assumes the number of sources to be comparable to (if not much greater than) the number of recipients, while multicast protocols are often devised to satisfy a small set of sources communicating with a large set of recipients.

As a consequence of these differences, it is not practical to generalize IP multicast routing algorithms to route events in a content-based publish-subscribe system. For similar reasons, it is hard to implement a content-based publish-subscribe system on top of an existing IP multicast protocol. This issue is discussed in detail in [12], where several alternatives are compared.

The term "group communication" identifies a body of research whose goal is to provide mechanisms for reliable communication among a group of possibly remote processes, and in addition to guarantee some degree of ordering and atomicity [5] of message delivery. Under this umbrella fall reliable multicast protocols [10, 11] as well as more complex systems such as Isis [2] and Horus [20].

In contrast to these goals, the main purpose of content-based publish-subscribe middleware is to distribute messages (i.e., events) to all the interested parties based on their content and to do so in a scalable and efficient way. This difference has a strong impact on the underlying algorithms and mechanisms adopted. For example, the implementation of most group communication systems distributes information by using either point-to-point connections from each source to each of the group members or IP multicast. Both of these approaches have drawbacks when applied to content-based publish-subscribe middleware. The former because it does not scale well, and the latter because, as already mentioned, it is very difficult to use IP multicast to efficiently route events based on content.

8. CONCLUSIONS

Currently available content-based publish-subscribe systems adopting a distributed event dispatcher do not provide any special mechanism to support the reconfiguration of the topology of the dispatching infrastructure to cope with changes in the underlying physical network. Solutions available in the literature at best mention a strawman solution whose simplicity is often outweighed by its inefficiency, since it involves areas that should not be affected by reconfiguration. In [14] we already tackled this problem by describing an algorithm that improves the strawman solution and can be applied to every scenario of reconfiguration.

In this work, we pushed optimization to an extreme by first identifying the minimal portion of the dispatching tree that is affected by the reconfiguration, then exploiting this notion to develop an algorithm that further reduces the traffic involved in the reconfiguration. Simulations show a reduction in the overhead of reconfiguration up to 76% with respect to the strawman solution. These results come at the price of a partial reduction in the general applicability of the algorithm.

Our current research efforts are geared towards two dimensions. On one hand, as we mentioned in Section 3, we are investigating solutions to the other two problems introduced by reconfiguration, namely, the reconfiguration of the overlay network and the minimization of event loss. On the other hand, we are developing a prototype of a publish-subscribe middleware supporting reconfiguration. This prototype is structured as a framework, in the object-oriented sense, and allows to specialize the middleware along several dimensions, in particular including the strategy to deal with reconfiguration. This prototype will ultimately enable us to validate the suitability of our approaches to reconfiguration in real scenarios.

9. REFERENCES

[1] G. Banavar et al. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proc. of the 19th Int. Conf. on Distributed Computing Systems*, 1999.

[2] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 9(12):36–53, December 1993.

[3] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[4] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, Aug. 2001.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[6] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe. Technical report, Politecnico di Milano, 2003. Available at `www.elet.polimi.it/~picco`.

[7] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, Sept. 2001.

[8] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proc. of the 19th IEEE Int. Conf. on Distributed Computing Systems—Middleware Workshop*, 1999.

[9] D. Heimbigner. Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality. In *Proc. of SAC*, 2001.

[10] B. Levine and J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *ACM Multimedia Systems Journal*, 6(5):334–348, August 1998.

[11] K. Obraczka. Multicast transport protocols: a survey and taxonomy. *IEEE Communications Magazine*, 36(1):94–102, January 1998.

[12] L. Opyrchal et al. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proc. of Middleware 2000*, 2000.

[13] A. Oram, editor. *Peer-to-Peer—Harnessing the Power of Disruptive Technologies.* O'Reilly, 2001.

[14] G. P. Picco, G. Cugola, and A. Murphy. Efficient content-based event dispatching in presence of topological reconfigurations. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems (ICDCS03)*, Providence (RI, USA), May 2003. To appear. Available at `www.elet.polimi.it/~picco`.

[15] D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6th European Software Engineering Conf. held jointly with the 5th Symp. on the Foundations of Software Engineering (ESEC/FSE97)*, LNCS 1301, Zurich (Switzerland), Sept. 1997. Springer.

[16] E. Royer and C. Perkins. Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol. In *Proc. of the 5th Int. Conf. on Mobile Computing and Networking (MobiCom99)*, pages 207–218, Seattle, WA, USA, August 1999.

[17] B. Segall et al. Content Based Routing with Elvin4. In *Proc. of AUUG2K*, Canberra, Australia, June 2000.

[18] P. Sutton, R. Arkins, and B. Segall. Supporting Disconnectedness—Transparent Information Delivery for Mobile and Invisible Computing. In *Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid*, May 2001.

[19] Univ. of California, Irvine. *Workshop on Internet Scale Event Notification*, July 1998. `http://www1.ics.uci.edu/IRUS/twist/wisen98/`.

[20] R. van Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[21] A. Varga. OMNeT++ Web page, 2003. `www.hit.bme.hu/phd/vargaa/omnetpp.htm`.

[22] M. Wray and R. Hawkes. Distributed Virtual Environments and VRML: an Event-based Architecture. In *Proc. of the 7th Int. WWW Conf.*, Brisbane, Australia, 1998.

[23] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for Internet-scale event services. In *Proc. of the 8th Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1999.

APPENDIX

Formalization

This appendix contains the formalization of the algorithms described in Section 4.1 and in Section 4.3. The algorithms are expressed as a set of actions, which are triggered by the conditions outlined in Figure 9. Figures 10 and 8 define the notation used in the rest of the formalization as well as the data structures referred to by the actions.

Each action definition is described with respect to the dispatcher performing the action, generically referred to as self. For example, if an event arrives at node self, all variables referenced in the action eventReceived are either local to self or are bound as parameters of the action. When an action is selected for execution, it executes as a single atomic step.

The formalization is structured as follows. Figure 12 depicts the actions which characterize the behavior of a standard publish-subscribe system. Figure 13 augments the standard processing by including the two actions necessary for implementing the strawman solution described in Section 4.1.

Finally, Figure 14 contains the actions responsible for the operation of the algorithm presented in this paper. In particular, substituteLinkTo is the action triggered when a link is being replaced with another. It is worth noting that this figure also contains a modified version of the action unsubscriptionReceived which takes into account the ignored table I needed by our solution.

Each dispatcher maintains a subscription table defined as follows:

$$S \triangleq \{(n,p), n \in N, p \in P\}$$

The reconfiguration process uses an unsubscription ignore table defined as follows:

$$I \triangleq \{(d,p,rId), d \in D, p \in P, rId \in ID\}$$

Where the symbols have the following meanings

d	the host whose unsubscription messages have to be ignored.
p	the pattern for which unsubscriptions have to be ignored.
rId	the identifier of the reconfiguration process for which the unsubscriptions should be ignored.

The reconciliation of subscriptions accross the new link uses the following structure to store P_{add}.

$$\mathsf{storedP_{add}} \triangleq \{(rId, P_{add}), rId \in ID\}$$

where Padd is a set of event patterns.

Figure 8: Tables maintained by each dispatcher.

The following predicates express the events which can trigger the rules.

substituteLinkTo(d, RP)	the link to dispatcher d has been replaced with another, with RP as the corresponding reconfiguration path
removeLinkTo(d)	the link to dispatcher d has been removed without adding a new link
addLinkTo(d)	a link to dispatcher d has been added
xyzReceived(n)	the message xyz has been received from node n

Figure 9: Conditions for action activation.

Domains

P	the set of possible event patterns
D	the set of neighboring dispatchers in the dispatching tree for the current dispatcher
C	the set of clients connected to the current dispatcher
$N = D \cup C$	the set of nodes (clients and dispatchers)
ID	the set of possible reconfiguration identifiers

Variables

Unless otherwise specified the following variables are defined as belonging to the following domains:

A dispatcher	$d, d', \ldots$	$\in D$
A node	$n, n', \ldots$	$\in N$
A pattern	$p, p', \ldots$	$\in P$
A reconfiguration ID	rId	$\in ID$

In each action, self $\in D$ denotes the current dispatcher, i.e. the one which is executing the action.

Reconfiguration Path

RP	denotes the whole reconfiguration path
$head$	denotes the head path
$newLink$	denotes the new link
$tail$	denotes the tail path

Messages

The messages that dispatchers and clients can exchange belong to the following types:

SUB(p)	a subscription message
UNSUB(p)	an unsubscription message

The messages that only dispatchers can exchange belong to the following types:

RECMSG$(rId, P_{\mathsf{add}}, P_{\mathsf{del}}, RP)$	the reconfiguration message.
CTRLMSG(rId)	the control message
FLUSHMSG(rId, RP)	the flush message

Where the parameters have the following meanings:

p	an event pattern
rId	the unique identifier of the reconfiguration process
P_{add}	the set of patterns whose subscriptions need to be restored along the head of the reconfiguration path
P_{del}	the set of patterns whose subscription need to be deleted from the head of the reconfiguration path
RP	the reconfiguration path

Functions

newId()	returns a fresh identifier
send(n, msg)	sends a message msg to a client or dispatcher n

The following functions applied to a sequence of dispatchers Seq return respectively

first(Seq)	the first dispatcher in the sequence
last(Seq)	the last dispatcher in the sequence
next(Seq)	the dispatcher following self in the sequence
prev(Seq)	the dispatcher preceding self in the sequence

Figure 10: Definitions used in the formalization.

```
//Process unsubscription from node n
processUnsubFrom($n, p$) $\equiv$
  $S \leftarrow S - \{(n,p)\}$ //Unsubscribe n from pattern p
  if $\neg\exists n'((n',p) \in S)$ then
    //No more subscribers, send pattern to all neighbors
    $\forall d | d \neq n \bullet$ send($d$, UNSUB($p$))
  else if $\exists!\ n'((n',p) \in S)$ then
    if $(n' \in D)$ then
      //Only one subscriber left, send pattern to it
      send($n'$, UNSUB($p$))

//Send subscriptions in the subscription table to a new neighbor
sendSubsToNewLink($d$) $\equiv$
  $P_{\mathsf{add}} \leftarrow \{p : \exists n(n \neq d \wedge (n,p) \in S)\}$
  $\forall p \in P_{\mathsf{add}} \bullet$ send($d$, SUB($p$))

//Update subscription table during head-path reconfiguration
updateSubTable($P_{\mathsf{add}}, d_{\mathsf{next}}, P_{\mathsf{del}}, d_{\mathsf{prev}}$) $\equiv$
  $S \leftarrow S \cup \{(d_{\mathsf{next}}, p) : p \in P_{\mathsf{add}}\}$
  $S \leftarrow S - \{(d_{\mathsf{prev}}, p) : p \in P_{\mathsf{del}}\}$
```

Figure 11: Auxiliary macro definitions.

```
//A subscription is received from a neighboring node n
subscriptionReceived($n$, SUB($p$))
  if $(n,p) \notin S$ then
    //n is not subscribed to p
    $S \leftarrow S \cup \{(n,p)\}$ //subscribe n to pattern p
    if $\neg\exists n'((n',p) \in S)$ then
      //First subscription received by self for pattern p
      $\forall d | d \neq n \bullet$ send($d$, SUB($p$)) //send pattern to all neighbors
    else if $\exists! n'((n',p) \in S)$ then
      if $(n' \in D)$ then
        //Second subscription received by self for p
        send($n'$, SUB($p$)) //Send pattern to first subscriber

//An unsubscription is received from a neighboring node n
unsubscriptionReceived($n$, UNSUB($p$))
  processUnsubFrom($n, p$)

//Process event from a neighboring dispatcher or a client
eventReceived($n$, EVENT)
  //send event to all matching subscribers
  $\forall n' | (\exists p((n',p) \in S \wedge$ match(EVENT, $p$))) $\bullet$ send($n'$, EVENT)
```

Figure 12: Actions for subscription and event processing in a conventional subscription forwarding scheme.

```
//A link to dispatcher d has been added
addLinkTo($d$)
  $D \leftarrow D \cup \{d\}$
  sendSubsToNewLink($d$)

//The link to dispatcher d has been removed
removeLinkTo($d$)
  $P_d \leftarrow \{p : ((d,p) \in S)\}$
  $\forall p \in P_d$(processUnsubFrom($d, p$))
  $D \leftarrow D - \{d\}$
```

Figure 13: Actions for handling link insertion and removal. These actions are used by any of the solutions in the case where a vanished link cannot be substituted with another.

```
//An unsubscription is received from a neighboring node n
unsubscriptionReceived($n$, UNSUB($p$))
  if $(n,p) \in S \wedge \neg\exists rId((n,p,rId) \in I)$ then
    //n is subscribed to p and it is not in the ignore table for p
    processUnsubFrom($n, p$)

//External operation to replace a link to dispatcher d
substituteLinkTo($d, RP$)
  // the set of patterns to which d was subscribed along the old
  link
  $P_{\mathsf{add}} \leftarrow \{p : (d,p) \in S\}$
  $rId \leftarrow$ newId() //generate new reconfiguration identifier
  $D \leftarrow D - \{d\}$
  //emulate reconfiguration message from d to self
  recmsgReceived($d$, RECMSG($rId, P_{\mathsf{add}}, P_{\mathsf{add}}, RP$))

//Receive a control message from a neighboring dispatcher d
ctrlmsgReceived($d$, CTRLMSG($rId$))
  //Remove d from the ignore table
  $I \leftarrow I - \{(d,p,rId) : (d,p,rId) \in I\}$

//Receive a flush message from a neighboring dispatcher d
flushmsgReceived($d$, FLUSHMSG($rId, RP$))
  //Check if this dispatcher is waiting for a flush message
  if $\exists P_{\mathsf{add}}((rId, P_{\mathsf{add}}) \in \mathsf{storedP_{add}})$ then
    //Perform the reconciliation using the stored $P_{\mathsf{add}}$
    //Propagate subscriptions received before tree reconnection
    $\forall p | \exists n(n \neq \mathsf{prev}(RP) \wedge (n,p) \in S \wedge p \notin P_{\mathsf{add}}) \bullet$
      send(prev($RP$), SUB($p$))
    $\mathsf{storedP_{add}} \leftarrow \mathsf{storedP_{add}} - (rId, P_{\mathsf{add}})$
  else
    //Forward the flush message
    send(prev($RP$), FLUSHMSG($rId, RP$))

//Receive a RECMSG message from a neighboring dispatcher d
recmsgReceived($d$, RECMSG($rId, P_{\mathsf{add}}, P_{\mathsf{del}}, RP$))
  if self $\in$ head then
    $I \leftarrow I \cup \{(\mathsf{next}(RP), p, rId) : p \in P_{\mathsf{add}}\}$
    updateSubTable($P_{\mathsf{add}}$, next($RP$), $P_{\mathsf{del}}, d$)
    if self $\neq$ first($RP$) then send($d$, CTRLMSG($rId$))
    //the set of patterns to be removed along RP
    $P'_{\mathsf{del}} \leftarrow P_{\mathsf{del}} - \{p : \exists n'(n' \neq \mathsf{next}(RP) \wedge (n',p) \in S)\}$
    send(next($RP$), RECMSG($rId, P_{\mathsf{add}}, P'_{\mathsf{del}}, RP$))
  if self = first($newLink$) then
    $D \leftarrow D \cup \{\mathsf{next}(RP)\}$
    updateSubTable($P_{\mathsf{add}}$, next($RP$), $P_{\mathsf{del}}, d$)
    //if self is the first node in the RP, do not send ctrl msg
    if self $\neq$ first($RP$) then send($d$, CTRLMSG($rId$))
    sendSubsToNewLink(next($RP$))
    send(next($RP$), RECMSG($rId, P_{\mathsf{add}}, \emptyset, RP$))
  if self = last($newLink$) then
    $D \leftarrow D \cup \{d\}$
    //If this is the last node of the RP, stop propagating RECMSG
    if self $\neq$ last($RP$) then
      send(next($RP$), RECMSG($rId, \emptyset, \emptyset, RP$))
    //propagate unsubscriptions received
    //before tree reconnection
    $\forall p \in P_{\mathsf{add}} | \neg\exists n(n \neq d \wedge (n,p) \in S) \bullet$ send($d$, UNSUB($p$))
    if self $\neq$ last($RP$) then
      //Store $P_{\mathsf{add}}$ so that it will be used later
      //for the reconciliation
      $\mathsf{storedP_{add}} \leftarrow \mathsf{storedP_{add}} \cup (rId, P_{\mathsf{add}})$
  if self $\in$ tail $\wedge$ self $\neq$ last($RP$) then
    send(next($RP$), RECMSG($rId, \emptyset, \emptyset, RP$))
  if self = last($RP$) then
    //First unsubscribe
    removeLinkTo(first($RP$))
    if self $\neq$ last($newLink$) then
      //Then generate the flush message
      send(prev($RP$), FLUSHMSG($rId, RP$))
    else
      //Perform the reconciliation
      $\forall p | \exists n(n \neq \mathsf{prev}(RP) \wedge (n,p) \in S \wedge p \notin P_{\mathsf{add}}) \bullet$
        send(prev($RP$), SUB($p$))
```

Figure 14: Actions for dealing with link substitution in our approach.