Minimizing the Reconfiguration Overhead in Content-Based Publish-Subscribe

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

[1] Dip. di Elettronica e Informazione, Politecnico di Milano, Italy
{cugola, frey, picco}@elet.polimi.it
[2] Dept. of Computer Science, Univ. of Rochester, NY, USA
murphy@cs.rochester.edu

ABSTRACT
The publish-subscribe model provides strong decoupling among the components of a distributed application. This 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 which impact the topology of the dispatching infrastructure. This paper presents a description and analysis of a novel algorithm to deal with this kind of reconfiguration. The strength of this algorithm is its ability to minimize the portion of the system affected by the reconfiguration by exploiting a novel concept we refer to as the *reconfiguration path*. Simulations compare our approach with two others and show a significant reduction (up to 76%) in the overhead caused by reconfiguration.

Keywords
Publish-subscribe, middleware, reconfigurable systems

1. INTRODUCTION
Publish-subscribe middleware is enjoying increasing popularity. After the initial centralized implementations, commercial and academic efforts are bringing increased scalability by distributing the event dispatching infrastructure. The next major challenge is to allow distributed publish-subscribe to support reconfiguration in order to adapt to topological changes in the physical network. Such reconfiguration may come from explicit changes in a controlled environment, e.g., an enterprise environment whose system administrator adds new machines to cope with increased load. It may also arise in less controlled scenarios such as mobile environments, where the movement of hosts communicating through wireless links affects the physical network topology. The publish-subscribe middleware should tolerate such reconfigurations with minimal or no perturbation of its normal operation.

SAC'04, March 14-17, 2004, Nicosia, Cyprus

The majority of currently available publish-subscribe middleware ignores the issue of reconfiguration: only a few systems employ an inefficient strawman solution. Our earlier work [10] addressed reconfiguration with the goal of identifying a solution applicable to every possible reconfiguration scenario. The resulting algorithm is a compromise between a performance improvement over the strawman solution and applicability. However, in controlled reconfiguration scenarios, this approach generates more overhead than necessary. Our goal in this paper is then to reduce the overhead as much as possible, even at the cost of reducing the applicability of the resulting solution. The algorithm we devised, besides being useful by itself, also represents a "minimal" solution, and is thus applicable for evaluating the performance of other approaches.

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 the solution of [10]. The results show a significant overhead reduction, up to 76% with respect 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 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 assessing our algorithm. Section 6 discusses the implications of our approach on the reconfiguration scenarios. Finally, Section 7 discusses related efforts in the field and Section 8 ends the paper with brief concluding remarks.

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. Recently, many publish-subscribe middleware have become available, which differ along several dimensions [3, 5, 11].

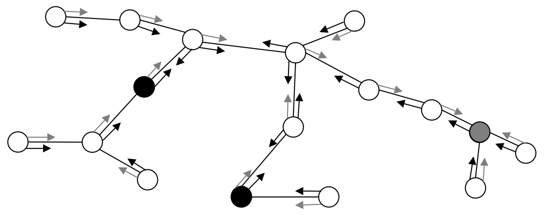

Figure 1: Subscription forwarding.

`In this paper we focus` on content-based publish-subscribe middleware adopting a distributed infrastructure to route events. Content-based systems enable clients to specify subscriptions using a pattern matching language (e.g., based on regular expressions). These systems provide much greater flexibility with respect to subject-based systems, which only allow matching against predefined classes of events. Nevertheless, the price is added complexity in the implementation. In the context of our problem, existing solutions developed for subject-based systems (e.g., using multicast) are not applicable, since a predefined notion of subject does not exist in content-based systems.

Our solution is developed by assuming that event routing is performed by using subscription forwarding [3] on an unrooted tree topology, as this choice covers the majority of existing systems. In a subscription forwarding scheme, 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 subscription forwarding of the same event pattern in the same direction[1]. 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.

Figure 1 shows a dispatching tree where some dispatchers (the black and gray ones) are subscribed to two different event patterns. The arrows represent the routes laid down according to these subscriptions, and reflect the content of the subscription tables of each dispatcher. 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]Other optimizations are possible, e.g., through "coverage" or aggregation of subscriptions, as in [3].

3. THE RECONFIGURATION PROBLEM

The reconfiguration problem 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 (un)subscriptions;
3. the minimization of event loss during reconfiguration.

The objective of the work we describe here 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 single unrooted 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. Our ongoing research activities are focused on addressing the other two problems. Specifically, we are investigating how existing overlay maintenance algorithms can be adapted to solve the first problem, and we already devised solutions to the third one based on epidemic algorithms [4].

Within this same framework, [10] tackled the second problem without making any assumptions about the sources of reconfiguration or the way the overlay network is kept connected. In this paper we adopt a different approach. Our goal is *i)* to push overhead optimization as far as possible and *ii)* to identify the implications on the tree maintenance layer, and consequently on its applicability to various reconfiguration scenarios. Essentially, by pushing optimization to an extreme we seek to investigate the whole spectrum of possibilities available to deal with reconfiguration.

4. DEALING WITH RECONFIGURATION

To solve the aforementioned problem we start by observing 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. To identify a single solution applicable in every situation, we focus on reconfigurations that involve the removal of a link and the insertion

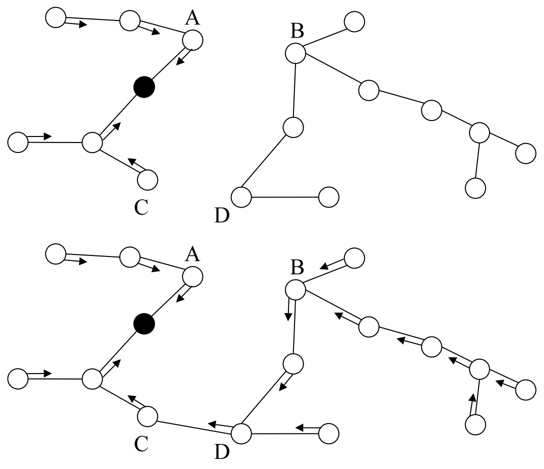

Figure 2: A dispatching tree during and after a reconfiguration performed using the strawman approach. Most subscriptions removed by a quickly propagating unsubscribe message in B's subtree have to be restored when the new link is established.

of a replacement that keeps the dispatching tree connected. Our motivation is that link substitution represents the fundamental building block for more complex reconfigurations. For instance, the disappearance of a dispatcher from a tree can be easily dealt with as a number of link substitutions reconnecting 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 that 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. In particular, the removal of a link can be addressed using unsubscriptions. In this case, neither of the end-points of the removed link is able to route events matching subscriptions issued by dispatchers on the other side of the tree; hence, each of them 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 the reconfiguration involves only either the insertion or the removal of a link, and it is actually adopted by some publish-subscribe middleware.

Nevertheless, in many other situations reconfiguration demands the replacement of a broken link with a new one, effectively changing the topology of the tree without changing the set of connected dispatchers. This can be dealt with by treating link substitution as a combination of the aforementioned link insertion and removal operations, e.g. as suggested in [3, 15], but the results are far from optimal. In fact, if the route reconfigurations caused by link removal and insertion 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, wasting a lot of messages and potentially causing far reaching and long lasting disruption of communication.

Figure 2 illustrates this concept on a simplified version of the dispatching tree of Figure 1. For simplicity, only subscriptions for a single event pattern p are shown, but in a real system the reconfiguration algorithm would process all event patterns in parallel. 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. The amount of disruption caused to the system depends on the message propagation speed.

The deficiency of this approach is that the reconfiguration messages reach areas of the dispatching tree that should not be affected at all by the reconfiguration. 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

From the perspective of event routing, the events that were intended to traverse the vanished link in order to reach the other part of the tree must be re-routed across the new link. This observation leads us to the definition of the *reconfiguration path* that contains the dispatchers connecting the old and new link. We define this path as the concatenation (without duplicates) of three sequences of dispatchers:

- the *head path* begins with the first end-point of the removed link (i.e., the end-point with the lowest identifier) and contains the sequence of dispatchers connecting it to the end-point of the new link that lies in the same subtree, which is included as the last node of the head path;
- the *new link* contains the end-point of the new link laying in the same subtree of the first end-point of the removed link, and the other end-point of the new link;
- the *tail path* begins with the other end-point of the new link, and contains the dispatchers connecting it to the second end-point of the removed link, inclusive.

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

The importance of the concept of reconfiguration path comes from the following consideration: *during the reconfiguration,*

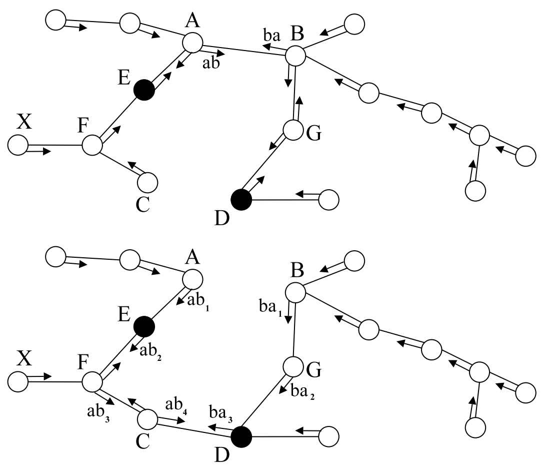

Figure 3: A dispatching tree before and after a reconfiguration.

a dispatcher that does not belong to the reconfiguration path does not experience a change in its subscription tables. Subscription information need only change on the dispatchers lying on the reconfiguration path.

4.3 Performing the Reconfiguration

Figure 3 provides an intuitive argument for the above statement. The subscription ab, which was exploiting the vanished link (A, B) to route events towards B's subtree, is removed by reconfiguration. Once the reconfiguration process has been completed, the routing information it provided is replaced by subscriptions ab_1, ab_2, ab_3, and ab_4. Similarly, the effect formerly achieved by ba is obtained by ba_1, ba_2, and ba_3 that, together with the subscriptions already present in A's subtree, enable events to reach the subscriber E.

To understand the details of the process it is worth making two observations. First, the subscriptions directed towards A's subtree only need to be added in B's subtree. This is because a subscriber's subtree always contains complete routing information to allow events to reach the subscriber from any of its dispatchers. Second, some of the subscriptions necessary to allow events to reach the other subtree may already be present due to other subscribers. In this example, in fact, only ab_2, ab_3, and ab_4 need to be added in A's subtree: ab_1 was already present to route events from A towards the subscriber E. In the other subtree, only ba_3 needs to be added towards A, since ba_1 and ba_2 were already present because of D.

These observations allow us to derive a general behavior: subscriptions replacing those from A to B are needed only on the head path. Similarly, subscriptions replacing those from B to A are needed only on the tail path.

To exploit the nature of the reconfiguration path, we devised an algorithm that propagates along dispatchers only in one direction, following the ordering imposed by the reconfiguration path and updating subscription tables along the way. The algorithm must correctly reconstruct the routing information (i.e., appropriately insert and delete subscriptions) in both directions, from A to B and vice versa. Thus, processing must be somehow different in the head and tail paths, and on the new link. Moreover, the normal operations of the publish-subscribe system are also being carried out during the reconfiguration, and the system must be left in a consistent state.

In the remainder of this section we present a reconfiguration algorithm that leverages off 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 (un)subscriptions issued during the reconfiguration; these are instead presented in Section 4.3.2.

4.3.1 Basic Operation of the Algorithm

This section steps through a single reconfiguration. Considerations for handling multiple, concurrent reconfigurations can be found in Section 6.

Starting the Reconfiguration. The reconfiguration process is started by the *initiator*, i.e., the first dispatcher on the reconfiguration path. 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. Looking at Figure 3, P_{add} enables the insertion of the missing ab_i subscriptions, 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 initialized with the same patterns: those belonging to subscriptions previously issued by the other end-point of the vanished link (ab in Figure 3). For each event pattern in P_{add}, a new entry is inserted in the initiator's subscription 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 identified a subscription by the other end-point of the removed link (arrows toward B in Figure 3) are deleted from the subscription table. In Figure 3, these actions cause respectively the insertion of ab_1 and the deletion of ab.

Reconfiguring the Head Path. After reconfiguration is completed at the initiator, the next step involves recomputing P_{del}. Generally, P_{del} must contain subscriptions used *only* to route events toward the removed link. Therefore if a dispatcher's subscription table contains a subscription to any dispatcher (or client) other than the next on the reconfiguration path, the subscription is removed from P_{del}.

Together P_{add}, P_{del}, and a list of the dispatcher identifiers in the reconfiguration path form the *reconfiguration message*, RECMSG, which is created at the initiator and propagated along the reconfiguration path. Each dispatcher receiving the RECMSG, performs the same operations originally performed by the initiator: update of the subscription table, re-computation of P_{del}, and propagation of a new RECMSG.

Reconciling Subscriptions Across the New Link. When the RECMSG reaches the first end-point of the new link (C in

Figure 3), the new link is physically available but it has not been logically "activated" in the tree. Therefore, because C is a member of the head path, it updates its subscription table as described earlier. Then, it activates the new link and sends to the second end-point a subscription message for each event pattern in its subscription table, followed by a RECMSG containing only P_{add}.

Completing the Reconfiguration. At this point, the subscriptions sent across the new link propagate throughout the second subtree normally. 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 subscriptions from G to B and 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 may be needed by other subscribers in the second subtree, and this information cannot be known by the RECMSG that travels along the tail path.Therefore, the second end-point of the new link propagates the RECMSG along the tail. Each dispatcher simply forwards the message, but when it reaches the last dispatcher on the reconfiguration path, this dispatcher behaves as if it had received unsubscription messages for each subscription toward the initiator in its subscription table (ba in Figure 3). These unsubscriptions propagate normally, eliminating only the superfluous subscriptions on the reconfiguration path.

4.3.2 Dealing with Concurrent (Un)Subscriptions

This section completes the algorithm description by addressing issues that arise when subscriptions and unsubscriptions are issued during the reconfiguration process.

Avoiding Race Conditions on the Head Path. To understand the first issue, we observe that the RECMSG flows along the head path in the opposite direction to normal subscription messages, yet part of its behavior is to add subscriptions. In other words, the RECMSG activates a subscription before the event recipient is aware of it. While this is normally acceptable, in the case where the recipient is already a subscriber and issues an unsubscription between the establishment of the subscription at the upstream dispatcher and the arrival of the RECMSG, the unsubscription has the effect of removing the subscription established by reconfiguration. Such behavior interrupts the propagation of events along the reconfiguration path to the second subtree.

The solution we adopt requires the upstream dispatcher to remember the subscriptions just added, and to delay the processing of unsubscriptions until the downstream dispatcher acknowledges the receipt of the RECMSG. This allows the upstream dispatcher to discern between an unsubscription which would disrupt event propagation along the reconfiguration path, and one that should instead be processed.

Reconciling Subscriptions Across the New Link. Prior to the arrival of the RECMSG at the second end-point of the new link, the second subtree is unaffected by the reconfiguration, and normal subscriptions and unsubscriptions can be issued without the possibility to reach the initiator's subtree. Therefore, the views of the two subtrees must be reconciled.

As for unsubscriptions on the second subtree, it is possible that the new subscriptions laid down on the initiator's subtree are no longer necessary. This can be determined by the second end-point of the new link (D in Figure 3) by comparing P_{add} against its own subscription table. For each subscription found in P_{add} but not in the local table, an unsubscription message is sent across the new link.

Similarly, there may be new subscriptions in the table that are not present in P_{add}. Immediately propagating such subscriptions upon receipt of the RECMSG may, however, lead to the propagation of subscriptions that are going to be removed by the unsubscription messages sent by the second end-point of the removed link. Therefore, we delay the propagation of these subscriptions until the second end-point of the new link receives notification from the last dispatcher that the reconfiguration process is complete. At this time, the second end-point of the new link has already processed any unsubscriptions generated during the reconfiguration. Therefore it can compare the copy of P_{add} it received earlier against its current subscription table and propagate only the necessary subscriptions.

5. SIMULATION RESULTS

In this section we assess our algorithm through simulation. The first goal is to verify whether the algorithm correctly restores routes upon reconfiguration. Our results show[2] that indeed event delivery returns to 100% after each reconfiguration. The other and primary goal, which we discuss thoroughly in the following, is to compare the reconfiguration overhead against other approaches. The baseline of our comparison is the strawman solution described in Section 4. In addition, we consider an enhancement, referred to with the oxymoron *"optimized strawman"*, which limits route disruption—and hence overhead—by appropriately performing unsubscriptions after subscriptions. Details can be found in [10].

The overhead is determined by the sum of three components: *i)* the (un)subscription messages being exchanged because of reconfiguration; *ii)* the event messages being misrouted along obsolete subscriptions; and *iii)* the additional messages required by our solution to limit the changes to the reconfiguration path. Message overhead is defined in terms of number of hops. Thus, for instance, a subscription issued by a dispatcher generates an overhead equal to the number of hops traveled by the subscription message.

Simulations were run by using the settings extensively described in [10]. Here, we only point out that we compare the algorithms in scenarios with low (20%) and high (80%) density of subscribers, and with low (1 publish/s) and high (50 publish/s) event publishing load per dispatcher. As in [10], data points are reported together with their Bezier interpolation to help visualize the overall trend.

Overall improvement. Figure 4 shows the percentage of im-

[2]The interested reader can find the corresponding charts and additional details in the full technical report [6].

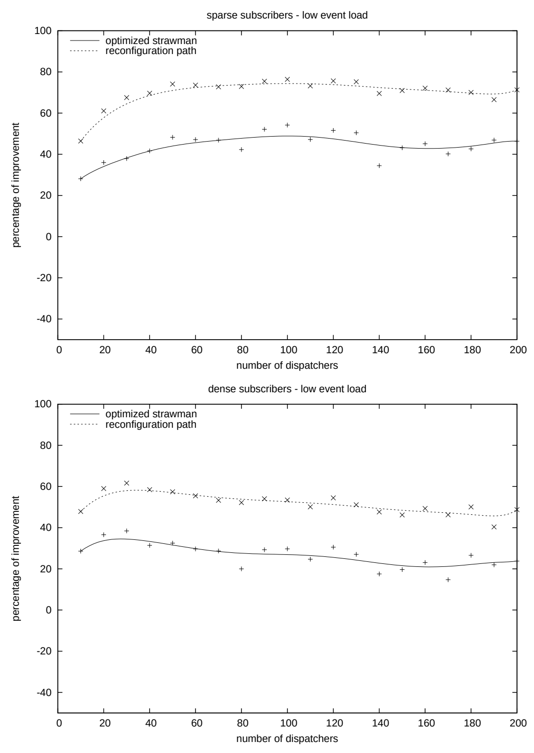

Figure 4: Improvement against the strawman solution with a low (top) and a high (bottom) density of subscribers.

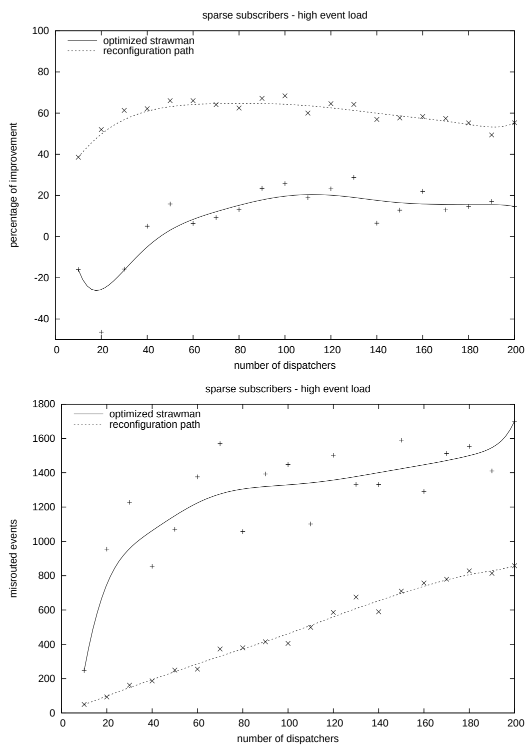

Figure 5: Improvement against the strawman solution (top) and number of misrouted events (bottom) in a scenario with low density of subscribers and high event load.

provement of both our solution and the optimized strawman with respect to the strawman solution, in a configuration with low publishing load. The charts confirm the intuition that both these solutions 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 in the other two cases. Moreover, in sparse trees (i.e., trees with few dispatchers) our approach improves more than the optimized strawman, since it limits the reconfiguration strictly to the reconfiguration path while the optimized strawman algorithm allows a small fraction of unnecessary subscriptions to travel farther.

To give a feel for the relevance of the percentage improvement, in a tree of 200 dispatchers a strawman reconfiguration generates an average of 3,500 messages (with a peak of 8,000) when the density of subscribers is high, and 5,300 (with a peak of 18,000) when it is low. Hence, even the improvement provided by the optimized strawman—15% in a dense tree—already leads to a significant overhead reduction, which is improved further (up to 40%) by our solution.

The top chart in Figure 5 shows how the overhead is affected by a low density of subscribers when the event load is high. Comparing this chart to the top one in Figure 4, it is evident that the improvement brought by both solutions is smaller under a high event load, due to the increased number of misrouted events.

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 reconfiguration. The solution that suffers most from this phenomenon is the optimized strawman, since it may maintain obsolete routes for a longer period of time due to its deferral of unsubscriptions. Our solution exhibits a lower overhead because the stale subscriptions that can cause the propagation of misrouted events are situated only along the reconfiguration path. The bottom chart in Figure 5 shows the difference in the number of misrouted events in a scenario with high event load.

Going back to the top chart in Figure 5 and the top chart in Figure 4, 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 simulation results show 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 [10]. 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.

The algorithm we 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, concurrent reconfigurations can be easily taken into account by distinguishing them through a reconfiguration identifier, as long as their reconfiguration paths do not overlap, i.e., they do not share a link. In this case, in fact, the corresponding reconfiguration processes would interfere.

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 is currently deployed, i.e., large enterprise networks.

Another scenario that recently became popular and that can exploit our approach is provided by peer-to-peer applications [9]. 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. This is where our approach can be applied. In fact, as observed also by other researchers [8], 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 fits the constraints identified above since connection and disconnection of peers is usually 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 the control of the application and it is difficult to avoid concurrent, overlapping reconfigurations. While our approach currently does not solve these issues, we are investigating alternative mechanisms exploiting the novel notion of reconfiguration path introduced by this paper. On the other hand, it is worth noting that the solution we presented in [10] 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 [10] still brings significant improvements but for arbitrary reconfigurations. Clearly, the choice of which approach is best is tied to the nature of the deployment environment.

7. RELATED WORK

Most publish-subscribe middleware target local area networks and adopt a centralized dispatcher. Recently, the problem of wide-area event notification attracted the attention of researchers and systems exploiting a distributed dispatcher became available, e.g. ([5, 3, 7, 14, 1, 12]). To the best of our knowledge, none of them provide any mechanism to support the reconfiguration addressed by this paper.

The closest match is the work on Siena [3] and the system described in [15]. 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 [5] 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 [13]. Finally, IBM Gryphon [1] and Microsoft Herald [2] 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.

8. CONCLUSIONS

The problem of dealing with topological reconfiguration in publish-subscribe middleware has been brought only recently to the attention of the research community. Existing solutions are based on a strawman approach whose simplicity is often outweighed by its inefficiency. The only example of an enhanced solution is described in [10]. In this work, we wanted to push the overhead reduction further and, at the same time, investigate the required assumptions about the management of the underlying topology. Our solution limits reconfiguration to a well defined path involving the topological change, which however requires greater knowledge about the topology. In scenarios where these assumptions can be met, simulations show that our approach enables a remarkable 76% overhead reduction against the strawman solution, and significant improvements over the one described in [10].

Acknowledgements. The work described in this paper was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, and by the National Research Council (CNR) under the IS-MANET project.

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 Dist. Computing Systems*, 1999.

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

[3] 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.

[4] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proc. of the*

24th Int. Conf. on Distributed Computing Systems, 2004. To appear.

[5] 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.

[6] G. Cugola, D. Frey, A. Murphy, and G. Picco. Minimizing the Reconfiguration Overhead in Content-Based Publish-Subscribe. Technical report, Politecnico di Milano, 2003. Available at `www.elet.polimi.it/~picco`.

[7] 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 Wkshp.*, 1999.

[8] D. Heimbigner. Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality. In *Proc. of the ACM Symp. on Applied Computing*, 2001.

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

[10] G. Picco, G. Cugola, and A. Murphy. Efficient Content-Based Event Dispatching in Presence of Topological Reconfiguration. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems*, May 2003.

[11] 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*, LNCS 1301, Zurich (Switzerland), 1997. Springer.

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

[13] 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.

[14] 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.

[15] H. Yu, D. Estrin, and R. Govindan. A hierarchical Proxy Architecture for Internet-Scale Event Services. In *Proc. of the 8th Int. Wkshps. on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1999.