Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Middleware

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

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

[2] Dept. of Computer Science, University of Rochester,
P.O. Box 270226, Rochester, NY 14627, USA
E-mail: `murphy@cs.rochester.edu`

Abstract. Publish-subscribe middleware allows the components of a distributed application to subscribe for event notifications and provides the infrastructure enabling event 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 work, we first describe two solutions available in the literature that constitute the extremes of the reconfiguration spectrum in terms of the number of nodes potentially affected by the reconfiguration. They differ essentially in the tradeoffs between simplicity and efficiency. Then, we introduce our contribution as a new algorithm that strikes a balance between the aforementioned solutions by tolerating frequent reconfigurations at the cost of moderate overhead.

1 Introduction

Publish-subscribe middleware are rapidly gaining popularity mostly because the asynchronous, implicit, multi-point, and peer-to-peer communication style they foster is well-suited for many 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 exploiting a distributed event dispatching architecture.

Beyond scalability, the next challenge for publish-subscribe middleware is dynamic reconfiguration of the topology of the distributed dispatching infrastructure. Companies are frequently undergoing administrative and organizational changes, and so is the logical and physical network enabling their information system. Mobility is increasingly becoming part of mainstream computing. Peer-to-peer networks are defining very fluid application-level networks for information sharing and dissemination. The very characteristics of the publish-subscribe *model*, most prominently the sharp decoupling between communication parties, make it amenable to these and other highly dynamic environments. However, this can be true in practice only if the publish-subscribe *system* is itself capable of dealing with reconfiguration. In particular, all the aforementioned sources of reconfiguration affect the topology of the event dispatching network, forcing the middleware to reconfigure its operations accordingly.

The vast majority of currently available publish-subscribe middleware has ignored this problem thus far. With the exception of a few systems adopting a very simple and inefficient solution, none of the proposals in the literature deal with dynamic reconfiguration. In [6], we tackled this problem for the first time by presenting an algorithm that minimizes the number of nodes involved in the reconfiguration. However, this algorithm is mostly suitable for environments where reconfiguration is somehow controlled, or in any case does not occur frequently. In this paper, we present instead a different algorithm that is designed expressly for highly dynamic environments, and that tolerates frequent reconfigurations at the cost of potentially incurring moderate overhead.

The paper is structured as follows. Section 2 provides a concise introduction to publish-subscribe middleware and a discussion about the possible sources of reconfiguration. Section 3 briefly describes a strawman solution to the problem of dynamic reconfiguration of publish-subscribe systems and the aforementioned solution delimiting reconfiguration. The comparison between the two provides the insight leading to the novel algorithm for highly dynamic environments that constitutes the main contribution of this work, and that is described in Section 4. Related research efforts are discussed in Section 5. Finally, Section 6 draws some conclusions and discusses future avenues of research.

2 Background and Motivation

In this section we provide an overview of publish-subscribe systems, together with a description of the reconfiguration scenarios that motivate our work.

2.1 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.

The communication and coordination model that results from this schema is inherently *asynchronous*; *multi-point*, because events are sent to all the interested components; *anonymous*, because the publisher need not know the identity of subscribers, and vice versa; *implicit*, because the set of event recipients is determined by the subscriptions, rather than being explicitly chosen by the sender; and *stateless*, because events do not persist in the system, rather they are sent only to those components that have subscribed before the event is published.

These characteristics result in a strong decoupling between event publishers and subscribers, which greatly reduces the effort required to modify the application architecture at run-time to cope with different kinds of changes in the external environment.

Given the potential of this paradigm, the last few years have seen the development of a large number of publish-subscribe middleware, 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 is applicable to both classes of systems but in this paper we assume a content-based subscription language, as this represents the most general and challenging case.

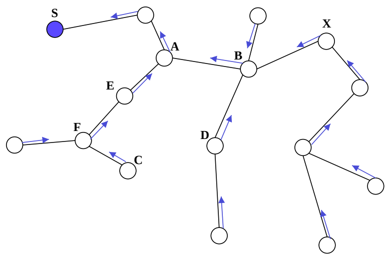

Fig. 1. Distributed publish-subscribe middleware. Clients have been omitted for clarity.

The architecture of the event dispatcher can be either centralized or distributed. In this paper, we focus on publish-subscribe middleware with a distributed event dispatcher. In such middleware (see Figure 1) a set of interconnected *dispatching servers*[2] cooperate in collecting subscriptions coming from clients and in routing events, with the goals of reducing network load and increasing scalability.

The systems exploiting a distributed dispatcher can be further classified according to the interconnection topology of dispatching servers, and the strategy exploited to route subscriptions and events. In this work we consider a subscription forwarding scheme on a tree topology, as this choice covers the majority of existing systems.

[1] For more detailed comparisons see [4, 5, 12].

[2] Unless otherwise stated, in the following we will refer to *dispatching servers* simply as *dispatchers*, although the latter term refers more precisely to the whole distributed component in charge of dispatching events, rather than to a specific server that is part of it.

In a subscription forwarding scheme [4], subscriptions are delivered to every dispatcher, along a single 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 event pattern is sent to the dispatcher the client is attached to. There, the event pattern representing the subscription is inserted in a subscription table, together with the identifier of the subscriber. Then, the subscription is propagated by the dispatcher, which now behaves as a subscriber with respect to the rest of the dispatching tree, to all of its neighboring dispatchers. In turn, they record the subscription and re-propagate it towards all their neighboring dispatchers, except for the one that sent it. This scheme is typically optimized by avoiding propagation of subscriptions to the same event pattern in the same direction[3]. The propagation of a subscription 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 a dispatcher (the dark one) is subscribed[4] 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.

2.2 Sources of Dynamic Reconfiguration

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 routes traversed by events in response to changes in the topology of the network of dispatchers, and to do this without interrupting the normal system operation.*

Triggers for such a reconfiguration are many, with the effect being the disappearance of one or more links between dispatchers, and possibly the appearance of new ones. A link can disappear either because it is being explicitly removed at the application layer, or because the underlying communication layers are no longer capable of ensuring communication between the two nodes.

The first case is clearly the most controlled one. As an example of this case, the publish-subscribe systems deployed in enterprise usually rely on a backbone of interconnected dispatchers. A system administrator may need to substitute

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

[4] More precisely, only clients can be subscribers. With some stretch of terminology, here and in the following we will say that a dispatcher is a subscriber if it has at least one client that is a subscriber.

one link with another to change the topology of the event dispatcher, e.g., to balance the traffic load or to adapt to a change in the underlying physical network. The result of such an operation should be an automatic reconfiguration of the distributed dispatcher to adapt event routes to the new topology.

Unfortunately, the sources of reconfiguration are not always under the control of applications. A dispatcher may become disconnected from one of its neighbors because the link connecting the two has failed. Mobile computing defines a scenario where this is particularly likely to happen. Mobility undermines the assumptions traditionally made in distributed systems by enabling the network topology to change dynamically as the mobile hosts move and yet remain connected through wireless links. This is brought to an extreme by mobile ad hoc networks (MANETs) [10], where the networking infrastructure is totally absent and physical links come and go according to the distance between hosts. In all these cases, lack of communication with a dispatcher results in the inability to route subscriptions and events towards it, due to the partitioning of the dispatching tree. A reconfiguration process is needed not only to restore the tree connectivity, but also to properly rearrange the routing information on the tree.

A somehow intermediate scenario is provided by peer-to-peer systems. In fact, the ability to perform scalable content-based event routing provided by distributed publish-subscribe middleware can be exploited to implement data sharing applications based on a peer-to-peer architecture. This idea has been exploited in PeerWare [7], a peer-to-peer middleware developed in the context of the EU project MOTION[5], and is also described in [9]. In this setting, each peer node plays the same role of a dispatcher in a publish-subscribe middleware, contributing to message routing. Consequently, the underlying routing mechanism must be able to cope with frequent changes of the topology of the peer network, determined by users (and hence peers) joining and leaving the peer-to-peer system.

3 Reconfiguration Extremes

In this paper, we focus on reconfigurations that involve the removal of a link and the insertion of a new one, thus keeping the dispatching tree connected. Issues of the loss of a dispatching node are more complex because the dispatching tree is partitioned into more than two pieces. We will consider this in future work. Simpler reconfigurations, involving only link removal or insertion, can be dealt with using plain subscriptions and unsubscriptions, as we describe later on.

The problem of dynamically reconfiguring a publish-subscribe system can then be seen as composed of three subproblems. The first problem is to manage the reconfiguration of the dispatching tree itself, retaining connectivity among dispatchers without creating loops. The second problem is to reconfigure the subscription information held by each dispatcher, keeping it consistent with the changes in the reconfigured tree and without interfering with the normal pro-

[5] IST-1999-11400, `www.motion.softeco.it`.

cessing of subscriptions and unsubscriptions. The third problem is to minimize the loss of events during the reconfiguration.

In this paper, we focus on correctly reconfiguring the subscription information, i.e., on the second of the aforementioned problems. We assume that the underlying tree is somehow reconfigured, and we tolerate (minor) event losses. The rationale for this choice lies in the fact that the consistency of the subscription information is key for the correct functioning of a publish-subscribe system, and hence also for limiting the number of events lost. Moreover, the algorithms for keeping the underlying tree connected strongly depend on the specific reconfiguration scenario, and in any case some existing solutions are likely to be adaptable, as we briefly discuss in Section 4.3. Also, by operating in a dynamic environment, the applications we consider must tradeoff some degree of reliable delivery. It is possible to extend the solution presented here to incorporate some fault tolerant techniques, but we leave this for future research.

Under these premises, a simple and reasonable way to compare the effectiveness of different reconfiguration algorithms is to consider the number of dispatchers involved in the reconfiguration, i.e., the number of dispatchers whose routing tables are changed during the reconfiguration process. Intuitively, the smaller this number the less the reconfiguration interferes with the system. Hence, not only is the overhead reduced, but so is the disruption of event routes, and consequently the likelihood of an event loss. If we base our comparison only on this value, two approaches represent the extremes of a wide spectrum: a straightforward, strawman algorithm that attacks the reconfiguration problem using the same strategy adopted when the tree of dispatchers must be split in two subtrees or when two subtrees must be joined, and a more efficient algorithm that minimizes the number of dispatchers involved. The remainder of this section describes these solutions and compares them. This comparison helps us gather some observations that motivate the need for a different approach when the target scenario exhibits high dynamicity. A description of an algorithm tailored for such environments is given in Section 4, which represents the main contribution of the paper.

3.1 A Strawman Approach

In principle, the removal of an existing link and the insertion of a new one can be carried out by using exclusively the primitives available in a publish-subscribe system.

The reconfiguration triggered by a link removal 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 on 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. The insertion of a new link triggers a similar process that uses subscriptions to reconfigure the routing.

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

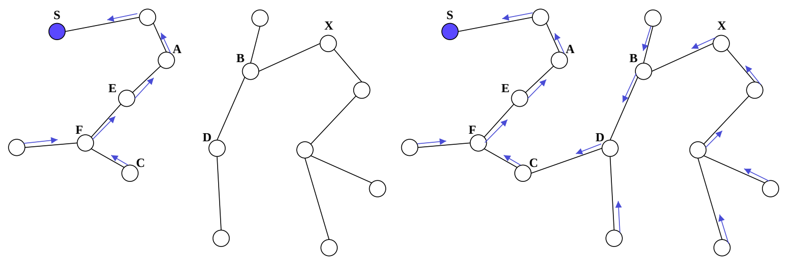

Fig. 2. The dispatching tree of Figure 1 during and after a reconfiguration performed using the strawman approach.

by some publish-subscribe middleware. On the other hand, when it is necessary to replace a link with a new one, thus effectively reconfiguring the topology of the tree while keeping the same set of nodes as dispatchers, this strategy leads to results that 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 then 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. According to the strawman mechanism, when the link between A and B is removed the two 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 in that tree. Nevertheless, shortly afterwards most of these subscriptions will be rebuilt by the reconfiguration process. The resulting reconfiguration of subscription information is not only inefficient, but it may greatly increase the loss of events.

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 the approach described in the next section.

3.2 A More Efficient Approach

To identify the minimal set of dispatchers affected by a link removal followed by a link insertion, we observe that each dispatcher routes events and subscriptions based on the local knowledge gathered from its neighbors. Similarly, its actions are limited to messages sent to its immediate neighbors. In other words, each dispatcher has knowledge only about its immediate “next hops”. In [6], these considerations lead to the definition of the *reconfiguration path* as the only portion of the dispatching tree affected by the reconfiguration. The reconfiguration path includes the two end-points of the removed link and all the dispatchers connected to them through the new link.

If we consider the example of reconfiguration in Figure 2, where the link (A, B) is being replaced with the link (C, D), the reconfiguration path is represented by (A, E, F, C, D, B). From the above considerations about the way subscription forwarding publish-subscribe systems work, it is easy to understand that dispatchers that do not belong to the reconfiguration path will not experience any change in their subscription tables. They will continue forwarding events the same way they were doing before. As an example, before reconfiguration (see Figure 1) dispatcher X was sending events to B, which was forwarding them to A to reach the subscriber S. After reconfiguration (see Figure 2), X continues sending events to B even though now B forwards them to D to reach the same subscriber S. X has no knowledge of the fact that B’s routing table has changed.

An algorithm that leverages off of this concept of reconfiguration path is described in [6]. Its processing starts from one of the two end-points of the removed link and uses a special source routed message that moves from dispatcher to dispatcher along the reconfiguration path, changing the routing tables according to the new topology.

3.3 Comparison

A first comparison of the two approaches described above, based only on the number of dispatchers involved in the reconfiguration, could lead to the conclusion that the second solution is always to be preferred over the first one.

Nevertheless, it turns out that the correctness of the strawman solution is not affected by multiple reconfigurations occurring in parallel. More precisely, if during a reconfiguration another link break occurs the two reconfigurations may proceed in parallel without influencing each other. Indeed, since the reconfiguration mechanism adopts only standard subscriptions and unsubscriptions and it does not affect the correct propagation of subscription and unsubscription messages, the overall reconfiguration process will complete correctly, independent from the number of link replacements involved[6].

Unfortunately, the same consideration does not hold for the algorithm described in [6] that, by rearranging the subscription information while unfolding along the reconfiguration path, strongly relies on its connectivity. As a result, this approach is quite sensitive to multiple reconfigurations. In particular, when different reconfiguration paths have one or more links in common or when an

[6] Here we assume that the process keeping the tree of dispatchers connected is capable of correctly handling multiple reconfigurations in parallel without introducing loops among the dispatchers and without resulting in partitioned trees.

additional link break does not allow a running reconfiguration process to complete as expected, special mechanisms must be put in place to guarantee the correctness of the overall reconfiguration process. Currently, these mechanisms are still under investigation, and hence the applicability of the approach covers only controlled environments where requests for multiple reconfigurations can be serialized and answered in sequence.

The above considerations motivate the need of a different algorithm for very dynamic environments such as MANETs or peer-to-peer networks, in which multiple reconfigurations occurring in parallel are more the rule than the exception. This algorithm should try to balance the performance, in terms of the set of dispatchers involved, of the solution described in [6] with the better resilience to multiple reconfigurations characterizing the strawman solution. The next section describes our proposal for such an algorithm.

4 Striking a Balance

To design a new algorithm for highly dynamic environments, we begin by observing that the drawbacks of the strawman algorithm described in Section 3.1 mainly result from the fact that the unsubscription process determined by a link removal and the subscription process taking care of a link insertion proceed completely in parallel, while some coordination would likely minimize the traffic. This consideration leads to the idea of identifying the impact of subscriptions and unsubscriptions on an already established tree to determine if some kind of synchronization could improve the performance of the strawman algorithm without sacrificing consistency when multiple link breaks occur in parallel.

4.1 Identifying the Tradeoffs

To describe the impact of subscriptions and unsubscriptions on a publish-subscribe system that adopts a subscription forwarding strategy, it is useful to classify dispatchers into subscribers, forwarders, and splitters[7]. For each event pattern p, a *subscriber* is a dispatcher that has at least one client subscribed to p. A *forwarder* is a dispatcher which is not a subscriber and whose routing table has a single entry tagged with p (i.e., graphically this means that it has a single outgoing arrow labelled with p). Finally, a *splitter* is either a dispatcher whose routing table has two or more entries tagged with p, or a subscriber.

With these definitions in mind, we can derive the following general rule for systems based on the subscription forwarding strategy described in Section 3.1: *a subscription issued by a client is propagated in the dispatching tree only up to the closest splitter, if it exists; to the whole tree, otherwise.* Clearly, in the special case where the new subscriber is also a splitter nothing happens.

[7] As already mentioned, these definitions do not take into account optimizations based on the notion of "coverage" among subscriptions, although they could be generalized to do so. Instead, the definitions are based on the usual optimization of avoiding to forward a subscription already present in the system.

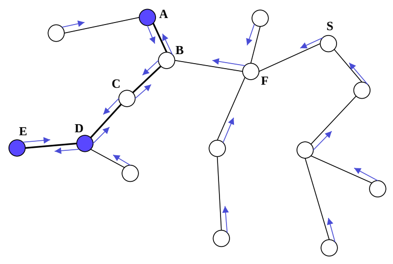

Fig. 3. A tree of dispatchers to show how new subscriptions propagate.

To understand this rule we observe that, for each event pattern p, there exists a minimal spanning tree containing all the dispatchers subscribed to p. For instance, in Figure 3 this minimal spanning tree is composed of dispatchers A, B, C, D, E. The routing tables of the dispatchers belonging to this subtree are organized in such a way that events matching p reaching one of them are forwarded to all the others. Moreover, the routing tables of all the other dispatchers route events matching p to this subtree but not vice versa, i.e., events reaching this subtree are not forwarded outside of it. Hence, we observe that the point of attachment for a new subscriber to such minimal subtree is constituted by the closest splitter. With reference to Figure 3, for the new subscriber S to join the subtree, only the routing tables of all the dispatchers along the path from S to the subtree (F and B in the figure) must be changed. A similar rule holds for unsubscriptions, which propagate up to the first splitter that remains such even after it has rearranged its subscription table by processing the unsubscription message.

From these rules it is possible to derive two considerations. First, the price that must be paid for adding a subscription is limited. In general, it does not involve a propagation along the entire tree but only along the route to the closest splitter, unless there are no subscribers. Second, the more splitters that exist the shorter the path that a subscription must follow. These considerations lead to the idea of an algorithm that behaves like the strawman one but performs the subscription step before unsubscribing. This way, the tree is kept "dense" of subscriptions, thus reducing the overhead caused by the propagation of subscriptions. It is true that this strategy may add subscriptions that must be removed immediately after, but in any case these subscriptions will propagate only up to the first splitter. Moreover, this solution has the beneficial side effect of minimizing the disruption of event routes, by minimizing the likelihood that a subscription is removed only to be restored shortly after. The next section describes such an algorithm in detail.

4.2 Rearranging Subscription Tables

In the following, we assume that the links connecting the dispatchers are FIFO and transport reliably (i.e., with no loss) 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.

The operation of the algorithm starts when a broken link between two dispatchers is detected. The actual processing is triggered by one of the two end-points, called the *initiator*, chosen according to some ordering criteria. The initiator starts a tree reconstruction process that tries to reconnect the tree without creating loops. For the moment, we gloss over the details of how the new link is identified and assume that this information becomes somehow available to the initiator after a given delay. We provide details about how this can be accomplished in reality at the end of this section. Here we focus on the processing needed to reconfigure the information for routing events over the reconnected tree.

The algorithm unfolds as follows:

1. When the end-points of a link detect that it is broken, they both start a timer T. In addition, the initiator starts the tree reconstruction process.
2. After the initiator (e.g., A in Figure 2) has determined the new link that needs to be established in order to reconnect the tree, it sends an OPENLINK message to the end-point of such link belonging to the same semitree as the initiator (C in Figure 2). The OPENLINK message is sent using a unicast channel that does not follow the dispatching tree, and must be acknowledged by the recipient using the same "out of band" channel. OPENLINK contains a reconfiguration identifier *recID*, which distinguishes it among multiple, concurrent reconfiguration processes. For each link, the value of *recID* is determined by the end-points when the link is first established successfully. Hence, the value of *recID* associated to the link is known to both end-points when reconfiguration actually occurs.
3. Upon receiving the OPENLINK message, the end-point of the new link belonging to the initiator semitree opens the new link and forwards the OPENLINK to the other end-point. Each end-point behaves as in a merge of the two semitrees, as described in Section 3.1, by exchanging their subscriptions over the new link, unless the old and new link share an end-point. In this case, subscriptions that exist only towards the other end-point of the new link are not forwarded. Moreover, immediately after the exchange, they start the propagation throughout their semitree of a FLUSH message containing the *recID* originally contained in OPENLINK.
4. At each dispatcher in each of the semitrees, subscriptions are propagated according to the normal processing. According to the discussion in the previous section, they propagate only up to the first splitter. Instead, the FLUSH message is broadcasted to all the dispatchers in the semitree[8].

[8] Ideally, the FLUSH message needs to propagate only along the reconfiguration path, up to the end-points of the vanished link. This can be accomplished if the tree reconstruction process provides information about the reconfiguration path. However, broadcasting along the whole tree is more resilient to concurrent reconfigurations.

5. If the FLUSH message reaches the end-points of the broken link or the timeout T expires, whichever occurs first, each of the end-points behaves as during a partition, by starting an unsubscription process for all subscriptions that came originally from the other end-point of the vanished link. In the case where the timeout expires, the corresponding *recID* is temporarily saved, to allow discarding of delayed FLUSH messages.

As we discussed before, this algorithm makes sure that subscriptions rerouting events through the new link are laid down *before* the obsolete subscriptions that served the only purpose to route events through the vanished link are removed. The OPENLINK message is essentially used to activate the new link and trigger the spreading of subscriptions between the semitrees. Instead, the FLUSH message is used to notify the end-points of the vanished link that it is now safe to remove unnecessary subscriptions. This property is ensured by the fact that the new subscriptions propagate ahead of the FLUSH message in FIFO channels. Essentially, OPENLINK triggers the portion of the reconfiguration taking care of merging the semitrees through the new link, while FLUSH triggers the partitioning of the semitrees across the vanishing link.

Clearly, in a highly dynamic environment connectivity may change during a reconfiguration, e.g., by causing multiple, concurrent link breaks. This does not constitute a problem if the reconfiguration paths determined by the breaks are not overlapping, in that reconfiguration can proceed independently. If instead the reconfiguration paths are overlapping, an additional link break may determine a temporary inability to communicate between the initiator and the end-point of the new link until a new tree reconstruction process has completed. Effectively, the second reconfiguration is "nested" in the first one, which cannot complete until the second is over. Besides increasing the overall time needed to recover from the first break, this situation may lead to a delay, if not a loss, of the FLUSH message.

This situation is handled by the last step of the algorithm, that performs the same *action* no matter whether a FLUSH message has been received, delayed or lost. Interestingly, the *effect* of such action in these situations is different, and is determined by the configuration of subscriptions that have already been laid out. When a FLUSH message is received, the corresponding new subscriptions are already setup correctly. Hence, the unsubscription process will remove only unnecessary subscriptions, along the reconfiguration path. On the other hand, if the timeout has expired two cases are possible. In the first case, no route reconnecting the tree exists. Hence, the unsubscriptions will rightfully propagate throughout the tree and possibly outside the reconfiguration path, up to the first splitter. In the case where the FLUSH message has been simply delayed, some overhead will result, depending on how fast the subscriptions ahead of the FLUSH message have travelled, with respect to the unsubscriptions triggered by the expiration of the timeout.

This latter aspect of the algorithm is controlled by the value of the timeout T. If T is too small, an unnecessary unsubscription process is likely to be triggered while the FLUSH is still on its way. On the other hand, if T is too large superfluous subscriptions are retained for a longer time, steering events towards dead branches of the tree. Essentially, the value of T must be chosen by evaluating a tradeoff between the responsiveness of reconfiguration and the bandwidth overhead caused by superfluous subscriptions. We are currently developing simulations to determine reasonable values for T and to investigate how they impact performance.

Notably, the reconfiguration described by this algorithm does not interfere with the normal processing of events and (un)subscriptions. In the solution we describe here, we are not trying to enforce any custom, source routed processing of messages like in the solution we described concisely in Section 3.2. Instead, we are relying on the standard processing that, by design, deals with the concurrent publishing of events and issuing of (un)subscriptions. We simply intervene in the timing when these operations are triggered to deal with reconfiguration. The only addition is the presence of a FLUSH message that, however, does not impact the normal processing.

Finally, our algorithm intuitively loses fewer events than the strawman solution. In fact, in the case where the FLUSH message is correctly received by the initiator, the routes for events are never disrupted. The only events lost are those that reached the end-points of the vanished link before the subscriptions exchanged through the new link. Instead, the strawman solution may lose events in areas potentially very far from the one where reconfiguration is occurring (i.e., from the reconfiguration path), since the uncoordinated propagation of subscriptions and unsubscriptions may temporarily remove routes. In the cases where the timeout expires and the unsubscription process is triggered, the amount of events lost is intuitively in between these two extremes.

4.3 Keeping the Tree Connected

Thus far, we focused only on how to update the routing information on the dispatching tree, without considering how a broken link is detected and a new route, involving a new link, is determined. In this section we hint at some ways of providing this functionality.

Detecting a Broken Link If the links between the nodes of the tree are actually mapped directly on physical communication links between the nodes, then detecting a link break can be dealt with in the same way as routing protocols for MANETs (e.g., DSR [2] or AODV [11]): essentially using MAC-level or application-level beaconing. A *beacon* is a packet that is periodically broadcasted with a time-to-live of 1, and hence reaches only the stations that are physically in communication range. When a station no longer detects a beacon[9] from another station, the link between the two can be considered broken. A similar approach can be adopted both in wired networks and when the logical link to be monitored does not map directly to a single physical link. In these cases a special

[9] Typically, a *k-out-of-n* policy is adopted, to avoid rapid fluctuations in connectivity.

point-to-point protocol, e.g., ICMP, must be used to implement the beaconing mechanism.

This proactive approach, however, constantly monitors the network. An alternative, lazier approach can detect link breakages only when a communication failure is notified at the application level, e.g., by an error returned while transmitting data on a socket. Clearly, this is possible only if the underlying transport protocol is reliable.

Replacing a Broken Link With a New Route After a broken link is detected, a new one must be found to reconnect the two partitioned subtrees without creating loops. The initiator must request a new route to its neighbors; new routes must be computed, possibly in a distributed way; they must be delivered back to the initiator, which will select one. A number of mechanisms can be used for this purpose.

For instance, it is reasonable to assume that each dispatcher maintains a cache of the network addresses of the dispatchers connected to its neighbors (i.e., each dispatcher has a partial visibility of the system topology). When a link vanishes, the initiator can send a message containing the list of dispatchers known to be part of the disconnected subtree, that gets propagated along the tree up to a certain number of hops. Each dispatcher receiving this message can then determine if it can reach one of the dispatchers on the list and how far it is, and send back a reply containing this information. The initiator uses the information to select the best route. The goal behind this process is clearly to keep the topology of the logical network of dispatchers as close as possible to the topology of the underlying physical network. In alternative, existing mechanisms for maintaining multicast trees can be used. For instance, for MANETs the strategy adopted by MAODV [13], heavily based on network-level broadcast and propagation of route requests, can be applied or adapted to our needs.

Thus far, we assumed that only a single link is added. This is reasonable in wired networks, where the routing infrastructure hides the details of communication between dispatchers. However, this may not hold true in a MANET or whenever the dispatching network is mapped directly on the network topology. In this case, one link is often not sufficient to reconnect the two partitioned subtrees, and additional intermediate nodes are needed. The new link can then be stretched into a sequence of nodes, whose end-points constitute the end-points of what we considered thus far as the new link.

5 Related Work

Most publish-subscribe middleware are targeted to local area networks and adopt a single, centralized dispatcher. In recent years, the problem of wide-area event notification has attracted the attention of researchers [16] and some systems have been presented, which adopt a distributed dispatcher, such as TIBCO's TIB/Rendezvous, Jedi [5], Siena [4], READY [8], Keryx [17], Gryphon [1], and Elvin4 [14] in its federated incarnation.

To the best of our knowledge, none of these systems provide any special mechanism to support the kind of reconfiguration proposed in this paper. Siena [4] and the system described in [18] adopt the strawman solution we describe later in Section 3.1 to allow subtrees of dispatchers to be merged or trees to be split. 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 runtime. A similar capability has been conceived also for Elvin [15], that supports mobile clients through a proxy server, although this feature is not included in the latest (4.0.3) release.

Finally, some research projects, like 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.

6 Conclusions and Future Work

Currently available publish-subscribe systems adopting a distributed event dispatcher do not provide any special mechanism to support the dynamic reconfiguration of the topology of the dispatching infrastructure to cope with changes in the external environment. Solutions available in the literature at best exploit a strawman solution whose simplicity is often outweighed by its inefficiency, since it involves areas that should not be affected by reconfiguration. Previous work by the authors has shown instead that there is a way to constrain reconfiguration, at the cost of increased complexity and poor tolerance to frequent topological changes.

In this work, we presented a solution that strikes a balance between these two reconfiguration extremes, by tolerating frequent reconfigurations at the cost of moderate overhead. Essentially, a mechanism is provided to ensure that the new routes caused by reconfiguration are laid down before the obsolete ones are removed. Besides optimizing the reconfiguration of routing information, this approach is also intuitively better at delivering events during reconfiguration.

Future work will investigate quantitatively the benefits of this solution against the other ones we described in this paper, using a simulation approach. Moreover, we will verify the feasibility of our approach by implementing a prototype and validating in the field.

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. J. Broch, D.B. Johnson, and D.A. Maltz. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. Internet Draft, October 1999.
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.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, August 2001.
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, September 2001.
6. G. Cugola, D. Frey, A. Murphy, and G.P. Picco. An algorithm for dynamic reconfiguration of publish-subscribe systems. Technical report, Politecnico di Milano, February 2002. Submitted for conference publication. Available at `www.elet.polimi.it/~picco`.
7. G. Cugola and G.P. Picco. Peerware: Core middleware support for peer-to-peer and mobile system. Technical report, Politecnico di Milano, November 2001. Submitted for conference publication. Available at `www.elet.polimi.it/~picco`.
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 the ACM Symposium on Applied Computing (SAC 2001)*, Las Vegas, NV, March 2001.
10. M.Corson, J.Macker, and G.Cinciarone. Internet-Based Mobile Ad Hoc Networking. *Internet Computing*, 3(4), 1999.
11. C.E. Perkins, E.M. Royer, and S.R. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet Draft, October 1999.
12. D.S. Rosenblum and A.L. 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), September 1997. Springer.
13. E.M. Royer and C.E. 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.
14. B. Segall et al. Content Based Routing with Elvin4. In *Proc. of AUUG2K*, Canberra, Australia, June 2000.
15. 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.
16. Univ. of California, Irvine. *WISEN, Workshop on Internet Scale Event Notification*, July 1998. `http://www1.ics.uci.edu/IRUS/twist/wisen98/`.
17. 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.
18. 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.