# Bringing Dynamic Reconfiguration into Publish-Subscribe Systems

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, University of Rochester, P.O. Box 270226, Rochester, NY, 14627, USA
murphy@cs.rochester.edu

## Abstract

*Publish-subscribe systems allow the components of a distributed application to subscribe for events, and provide an infrastructure to allow dynamic routing of such events from sources to subscribers. The resulting architecture yields a loose coupling among publishers and subscribers, making the model amenable to supporting dynamic applications in both wired and wireless environments. In contrast, however, most publish-subscribe systems are not themselves dynamic. Specifically, they usually do not provide any mechanism to reconfigure the dispatching infrastructure in response to the loss of a communication channel. Simple solutions have been proposed by others, exploiting the standard subscription and unsubscription mechanisms to update the event routing information within the dispatching infrastructure. In this paper, we show that such an approach is often inefficient, causing unnecessary updates to the subscription information throughout the dispatching network and potentially leading to the unnecessary loss of events during reconfiguration. After an analysis of the potential sources of reconfiguration, we propose an approach which constructs a reconfiguration path containing only those dispatchers whose subscription information may change in response to the topology change, thus limiting the impact of reconfiguration on the underlying event dispatching service. Further, we provide an analysis of the mechanisms necessary to detect the link failure and to construct the reconfiguration path in various application environments spanning from wired to mobile ad hoc networks.*

## 1 Introduction

In the last few years several distributed applications and middleware have been developed, which adopt a publish-subscribe architectural style. The popularity of the model can be explained by observing that the communication and coordination paradigm it adopts matches reasonably well the flexibility requirements of modern distributed applications.

In a publish-subscribe application, components interact through *event notifications* or simply *events*. These are messages issued by a *publisher* and received by every *subscriber* that expressed an interest in them. A special element of the architecture, the *event dispatcher*, is in charge of distributing events based on the declaration of interests made by subscribers.

This approach results in communication and coordination, which is inherently asynchronous and multi-point and in which components are loosely coupled. This potentially leads to the ability to add, remove, or even move components at run-time with a very limited impact on other parties. On the other hand, we said "potentially" since the large majority of currently available publish-subscribe middleware do not offer any special mechanism to explicitly support dynamic reconfiguration of the application architecture. Conversely, usually they assume that publishers and subscribers are stationary, that the pattern of communication is fixed, and that the underlying network topology does not change.

The lack of explicit mechanisms to support dynamic reconfiguration of the application architecture becomes even more evident in the presence of publish-subscribe middleware that provide a distributed implementation of the event dispatcher, built as a network of servers that cooperate to store subscriptions and route events. The need to support dynamic reconfiguration of the application architecture at run-time should suggest coupling this solution with a mechanism to change the topology of the distributed event dispatcher itself, in order to react to changes in the external environment. Unfortunately, with the exception of a limited number of systems, that adopt a very simple solution, none of the middleware proposed so far provide such a mechanism.

In this paper we present an approach to eliminate the above limitation, while reducing the impact of the reconfiguration. It enables the possibility of reconfiguring the network topology of a distributed event dispatcher by also minimizing the set of dispatching servers involved in the reconfiguration. As a secondary contribution of the paper, in Section 3 we analyze the possible causes of reconfiguration and classify them by identifying the most relevant scenarios of reconfiguration, while in Section 5 we describe the impact of these different scenarios on some aspects of our approach. In Section 6 we discuss some open issues, describe other potential areas of interest, and compare this work with state of the art technology, Finally, in Section 7 we draw some conclusions and discuss future avenues of research on the topic of this paper.

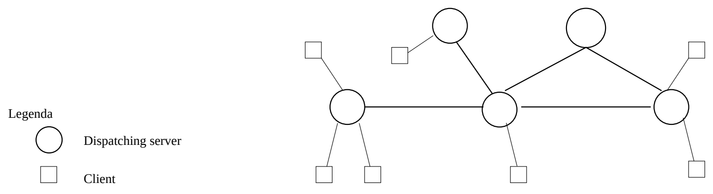

**Figure 1. Distributed publish-subscribe middleware.**

## 2 Publish-Subscribe Systems

As mentioned above, publish-subscribe applications are organized as a collection of autonomous components, which interact by *publishing* events and by *subscribing* to the classes of events they are interested in. A special element of the application, the *event dispatcher*, is in charge of collecting event subscriptions and distributing events to all the 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 identity of the sender is hidden from the receiver; implicit, because the set of recipients of each event is chosen implicitly, based on subscriptions, and cannot be changed by the sender; and stateless, because events are not persistently stored by the system, rather they are sent only to components that subscribe before the event is published.

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

Given the potential of this paradigm, in the last few years a large number of publish-subscribe middleware have been developed to reduce the effort required to implement distributed publish-subscribe applications. They differ in the format of events, ranging from simple tuples to typed objects; in the way subscriptions may be specified, which ranges from simple channels or subjects to very complex expressions (usually called *event patterns*) on the content of events; in the observation and notification models, which can be push or pull; and in the architecture of the event dispatcher they provide, which can be centralized or distributed with different topologies (cyclic or not). Moreover, where a distributed dispatcher is adopted, a further difference is in the way subscriptions and events are routed. Subscriptions, in fact, are used to define the routes that events have to follow to reach the interested parties. Several strategies may be adopted, that differ in the way subscriptions are propagated within the distributed dispatcher to find the best balance between the cost of subscribing and unsubscribing and the cost of distributing events [12, 5, 4].

In this paper, we focus our attention on publish-subscribe middleware that provide a distributed implementation of the event dispatcher, utually to target wide-area networks. In such middleware (see Figure 1) a set of interconnected *dispatching servers*[1] cooperate in collecting subscriptions coming from application components (often called *clients*) and in routing events, with the goal of reducing network load and increasing scalability.

The main limitation of such middleware is that usually they do not offer any explicit mechanism either to move clients from one place to another or to reconfigure the distributed dispatcher itself. In particular, they usually assume that publishers and subscribers are stationary, that the pattern of communication is rather fixed, and that the underlying network topology is fixed also. With the development of Jedi [5], we tried to overcome the first limitation by proposing a publish-subscribe middleware that allows clients to move from one place to another without loosing events while moving. In the rest of the paper we present an approach to overcome the second limitation by allowing changes in the topology of dispatching servers. In particular, we focus on publish-subscribe middleware whose event dispatcher is organized as an acyclic graph (i.e., a tree) of dispatching servers and on a topology change characterized by the loss of a single link which partitions the tree into two parts and the insertion of a single new link to reconnect those parts. Our approach minimizes the overhead of the reconfiguration process for this link substitution, limiting the effect on the underlying event distribution system. Handling such reconfiguration is fundamental in order to cope with various dynamic scenarios, the analysis of which is provided in the next section.

## 3 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 notion of dynamic reconfiguration we address can be defined informally as *the ability to rearrange the routes traversed by events in response to a change in the topology of the network of dispatchers, and to do this without interrupting the normal system operation.*

The causes that may trigger such a reconfiguration are many, and precisely characterizing them is crucial to define the requirements for the reconfiguration mechanisms. A link between two nodes of the dispatching graph can disappear for essentially two reasons: either because it is being explicitly removed at the application layer, or because the underlying

[1]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.

communication layers are no longer capable of ensuring communication between the two nodes.

The first case is clearly the most controlled one. The fact that the decision of removing a link is taken at the application layer eliminates the need of correctly detecting the link break and it results in the possibility of performing several optimizations, especially with respect to the way the new link is found, as discussed later. For instance, in an enterprise it is reasonable for a publish-subscribe system to rely on a rather stable backbone of interconnected dispatchers. In this situation, a link may be removed and substituted in a single step, through a proper application programming interface, by a system administrator who needs to change the topology of the event dispatcher, e.g., to optimize traffic, or to adapt to a change in the topology of 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 causes for reconfiguration are not always under the control of applications. We have all experienced the inability to reach a host on the Internet. This can be caused by a failure in the target node, or by a failure in some of the nodes or links that enable communication between our machine and the target one. In publish-subscribe systems like the ones we focus on, the inability of a dispatcher to contact one of its neighbors results in the inability to route subscriptions and events, since the tree of dispatchers is partitioned. It is of little relevance whether the failure, at the physical level, rests with the target node or with the communication infrastructure: the net effect is that the link is no longer available at the logical level, and a reconfiguration is needed. In this case, not only it is necessary to detect the link break, but also proper mechanisms must be put in place both to determine if a new link can substitute the disappeared one and to appropriately choose it if more than one link with the required characteristics exists.

Moreover, we observe that faults in the fixed network are usually temporary. Hence, the benefit of a reconfiguration must be weighed against the likelihood of another reconfiguration taking place right after, in case the link is restored quickly. This means that reconfiguration should be complemented by some policy that takes into account a fault model for the network at hand, whose treatment is nevertheless outside the scope of this paper.

Besides the enterprise scenario and conventional distributed systems in general, there are other scenarios that are rapidly gaining popularity and that intrinsically demand for a high degree of reconfiguration. Mobile computing, for instance, undermines the traditional assumptions made in distributed systems by enabling the network topology to change dynamically as the mobile hosts move, and yet retain connectivity through wireless links. This is brought to an extreme by mobile ad hoc networks (MANETs) [8], where the networking infrastructure is totally absent.

The aforementioned distinction between the changes induced by the application and those induced by the network holds. In fact, in the most constrained forms of mobility, nomadic users are not performing computation while moving, but only once they have settled in a specific location, which changes from time to time. In this scenario, typical of applications like impromptu meetings, conference and classroom settings, *planned disconnection* [7] is the norm. Connectivity does not cease abruptly, rather it is terminated by users, e.g., to save battery, or to move to a different location. On the other hand, in the most radical forms of mobility like those defined by MANETs the network is extremely fluid. Physical links come and go according to the movement of hosts, and hence remain outside the control of applications. Moreover, link failures are not necessarily temporary, as a link may never be re-established.

The peculiarity of the wireless scenario is in the probability of link breaks, which is usually much higher than in traditional networks, and in the different network protocols available. As we will explain in Section 5 this has a strong impact on the way some of the steps involved into the reconfiguration can be performed.

## 4 Dealing with Reconfiguration

Several strategies have been proposed in the literature to implement distributed publish-subscribe systems. Hence, we begin this section by illustrating the dispatching strategy we chose as the base for our work, and by stating additional assumptions we rely on. Then, we present a straightforward solution that provides reconfiguration by introducing minimal changes to the normal behavior of the publish-subscribe system. This solution, adopted by some of the available publish-subscribe systems, has a number of drawbacks that we tackle and overcome with a novel approach that is described in the rest of the section, and that constitutes the main contribution of the paper.

### 4.1 Reference Architecture

Our approach to reconfiguration assumes a dispatching strategy based on subscription forwarding [4]. In this scheme, subscriptions are delivered to all the dispatchers, and are used to establish the routes that are followed by published events.

When a client needs to subscribe for a given event pattern, it sends a subscription message to the dispatcher it is directly attached to. There, the subscription is inserted in a subscription table, together with the identifier of the subscriber. When an event matching the specified pattern reaches the dispatcher, a table lookup is performed to find the client subscribers for the event.

Subscriptions are propagated by the dispatcher, now behaving as a subscriber with respect to the rest of the dispatching tree. The dispatcher generates a subscription message towards all of its neighboring dispatchers, that in turn record it and re-propagate it towards all the neighboring dispatchers, except for the one that sent the message.

Propagation among dispatchers can be limited by examining the content of the subscription against the content of the subscription table, to avoid sending a subscription in areas of the tree where it has already been propagated. If a dispatcher receives a subscription $s$ never received previously by any of its neighbors, it propagates it in the usual way to all the dispatchers but the one that sent it. In turn, if the subscription table shows that $s$ has been received by only one dispatcher $d$, it means that routes for events have been setup in such a way that the current dispatcher is collecting events matching $s$ to forward them to $d$. Hence, the only change needed is to send $s$ also to $d$, so that events matching $s$ are sent also the opposite way, i.e., from $d$ to this dispatcher. Finally, if the subscription table contains entries for at least two neighboring dispatchers there is no need to propagate the subscription message any further[2].

Requests to unsubscribe from a given event pattern are handled and propagated in the same way, by removing entries in the subscription table rather than by adding them. The processing of events then relies on the dispatching tree annotated with subscription information. When a dispatcher receives an event, it first checks whether it should be delivered locally, i.e., to one of its clients, then it forwards it to those of its neighbors that appear in its subscription table paired with at least one event pattern matching the received event. A formalization of this behavior is provided in Appendix A.

## 4.2 Assumptions

In the following, we assume that the links connecting dispatchers are FIFO and that they transport reliably (i.e., with no loss) subscriptions, unsubscriptions, events, and other control messages. Both assumptions are typical of mainstream publish-subscribe systems.

As far as reconfiguration is concerned, in this work we limit ourselves to topological changes that involve either:

- the removal of a link between two dispatchers, thus effectively causing a partition in the tree;
- the insertion of a link between two dispatchers, thus allowing the merging of two previously disconnected trees, or the joining of a singleton dispatcher;
- the substitution of one link with another, thus effectively changing the topology of an existing tree.

These cases define a minimal granularity for reconfiguration. Our conjecture is that more complex reconfiguration patterns can be described as variations or combinations of the techniques used for dealing with link reconfiguration illustrated in this paper. More discussion about this latter topic is provided in Section 6.

[2]Other optimizations are possible, e.g., by defining a notion of "coverage" among subscriptions, or by aggregating them, like in [4]. However, their treatment has no immediate impact on the content of this paper, and hence will not be considered further.

Also, in this work we focus only on the tree topology for dispatchers. Dealing with trees reduces the complexity of the problem and allows us to concentrate on the key mechanisms needed for reconfiguration without having to deal with graph-related technicalities (e.g., avoiding loops). Moreover, this choice favors an immediate dissemination of our results into the state of the art, since the tree topology is the most popular among distributed publish-subscribe systems.

Finally, according to our definition reconfiguration is dynamic and occurs while the system is operating, i.e., while subscriptions and events are being distributed. Reconfiguration should then interfere as little as possible with normal operations: ideally, subscriptions and events should not be lost as a consequence of reconfiguration. In this work, we strike a balance between guarantees and efficiency by presenting an algorithm that guarantees no loss of subscriptions and unsubscriptions but admits loss of events, albeit for a very short period of time and only within the area directly involved in the reconfiguration. But, before illustrating our final solution, we need to take a look at a straightforward solution that is actually implemented by some of the available systems to deal with the partitioning of the tree, or with the merging of two trees. While this solution is effective in these cases, we will show that it is insufficient in the case of a link substitution, and propose a novel approach for this latter case.

## 4.3 A Straightforward 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 decoupling among the dispatchers in the tree.

For instance, 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 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 contained in its subscription table and belonging to subscriptions that route events towards the end-point on the other side of the disconnected link. Similarly, the insertion of a new link in the tree can be carried out in a dual way by having its two end-points exchange all the event patterns in their subscription tables, and propagating them as normal subscription messages in their respective subtrees.

The approach we just described 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-subscribe middleware to deal with merging or partitioning.

Nevertheless, as mentioned in Section 3, in many other situations it is desirable to substitute a link with a new one,

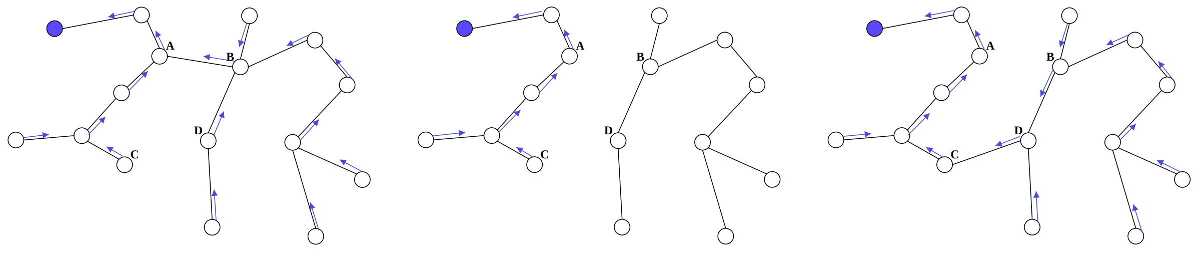

**Figure 2. A dispatching tree before, during, and after a reconfiguration performed using the straightforward approach.**

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 straightforward mechanism we described, e.g., as done in [18, 4], 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 then subsequently restored, thus wasting a lot of messages and potentially causing far reaching and long lasting disruption of communication.

Figure 2 shows a dispatching tree with a dispatcher (the dark one) subscribed[3] to a certain event pattern $p$, and arrows representing the routes laid down according to this subscription. An arrow directed from a dispatcher $x$ to a dispatcher $y$ means that $x$ will forward events matching $p$ to $y$, i.e., that the subscription table of $x$ contains an entry pairing $p$ with $y$. To avoid cluttering the figure, subscriptions are shown only for a single event pattern. According to the straightforward mechanism we examined, 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 for $p$ in that tree. Nevertheless, shortly afterwards most of these subscriptions will be rebuilt by the reconfiguration process.

Clearly, a lot of unnecessary communication takes place, wasting computation and communication resources. Even worse, routing of events is disrupted until the combined reconfiguration process is completed: for large networks, this may be too high a cost to pay. The problem can be alleviated by finding a way to sequence the two reconfiguration processes, to minimize the number of events lost. However, this is likely to increase significantly the time to complete a reconfiguration.

[3]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.

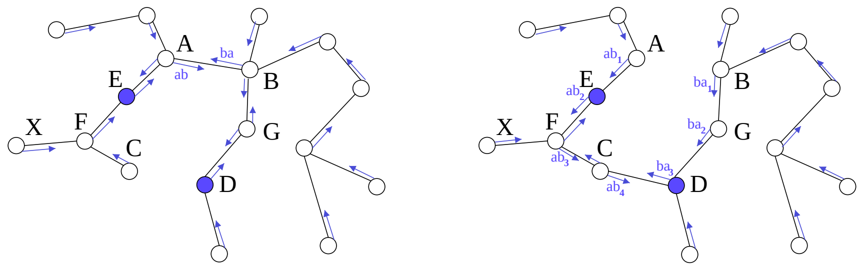

**Figure 3. A dispatching tree before and after a reconfiguration.**

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 very far from the ones directly involved in the topology change, and which should not be affected at all. This observation leads to the definition of a way to delimit the area involved in the reconfiguration, which is a key element of our approach.

## 4.4 Delimiting the Reconfiguration

Our goal is to find a new route along which to forward the events that formerly traversed the now vanished link to the dispatcher on the opposite side of the link. We refer to the path which reconnects these two end-points as the *reconfiguration path*. It includes exactly one link which was not in the previous tree and that, when taken as a whole, effectively replaces the one lost link. The 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 on 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. Section 5 will discuss further how the ordering between the end-points can be determined. Here, it suffices to observe that such an ordering, combined with the topological properties of the tree, make 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 $(G, B)$, hence 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 key 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 $E$. All $X$ knows is that $F$ subscribed to him to receive events matching $p$. The distributed knowledge disseminated in the dispatching tree will steer the event towards its destination, $E$ 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 by the dispatcher lying on the reconfiguration path. In Figure 3, for instance, $X$ will keep forwarding events to $F$ as usual.

## 4.5 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 through 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 $E$, allow 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 $B$; $ab_1$ 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 subscriber $D$.

By observing Figure 3 it can be noted 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. Following the same line of reasoning, it is also evident how the subscriptions that replace $ba$ are needed only on the tail path, i.e., from $B$ to $D$; after that point, events coming from $B$'s subtree are handled using the subscriptions already present in $A$'s subtree, again if no unsubscription occurred meanwhile.

This observation can be used to determine how to propagate subscriptions along the reconfiguration path. The reconfiguration process is indeed directional, since it propagates across dispatchers only in one direction by 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.

Moreover, redirecting subscriptions is not sufficient. While the reconfiguration is taking place, the normal operations of the publish-subscribe system are being carried out. Thus, the subscriptions and unsubscriptions that are being issued in portions of the tree where the reconfiguration has not yet arrived must be somehow taken into account by the process, as they might have an effect on those already laid on the reconfiguration path.

In the remainder of this section we present a reconfiguration process that leverages off of the definitions and observations made thus far. Additional details, including a formalization of the behavior of a publish-subscribe system extended with the reconfiguration capabilities described here, are provided in Appendix A.

**Starting the Reconfiguration** The reconfiguration process is started by the first of the end-points of the link being removed from the dispatching tree, that is also the first dispatcher on the reconfiguration path. We call this node the *initiator* of the reconfiguration. Again, here we gloss over the details of how the initiator is chosen, of how a link removal is detected, and of how a new link is chosen and the reconfiguration path made known to the initiator. We discuss these details in Section 5.

The initiator starts the reconfiguration by performing a table lookup to determine the event patterns belonging to subscriptions (e.g., $ab$ in Figure 3) previously performed by the other end-point of the vanished link. 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 $P_{add}$ is what enables the insertion of the $ab$ subscriptions that are missing on the path from $A$ to $C$, while $P_{del}$ is what 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 $A$.

**Reconfiguring the Head Path** At this point, the reconfiguration of the subscription table of the initiator is complete, and the reconfiguration can proceed along the reconfiguration path.

The propagation of reconfiguration involves computing a new set $P_{del}$, to be used by the next dispatcher on the reconfiguration path to update its tables. In fact, $P_{del}$ is in general 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 needed to route events towards different areas of $A$'s subtree should not be removed. However, this information can be computed locally by the initiator, and subsequently by each of the other dispatchers, by looking at their subscription table: patterns in $P_{del}$ for which a subscription exists towards a dispatcher other than the next one on the reconfiguration path are removed from $P_{del}$, which is then shrinking along 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, which is sent by the initiator to the dispatcher following it in the reconfiguration path. This message effectively starts the reconfiguration process, which unfolds along the head path with RECMSG being propagated along the head path by each dispatcher sitting on it. Each dispatcher, upon receiving RECMSG, performs the same operations originally performed by the initiator. First, the subscription table is updated by inserting new subscriptions that contain the patterns in $P_{add}$ and are directed towards the next dispatcher on the reconfiguration path, and by deleting the subscriptions for patterns in $P_{del}$ that are directed towards the previous dispatcher. Then, a new set $P_{del}$ is computed and a message RECMSG, modified to contain the new $P_{del}$, is propagated.

It is relevant to note how, across the head path, RECMSG propagates in the opposite direction with respect to the established subscriptions. This determines an unusual way of processing subscriptions. 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 in its subscription table. Instead, here the recipient of RECMSG becomes subscribed to events sent by the sender, which has already inserted the recipient in its subscription table. In practice, the recipient finds itself subscribed to certain events, and does not know that until it receives RECMSG. This is a possible cause of race conditions we deal with later on in this section.

**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, the new link is already physically available but it has not been logically "activated" in the tree by the reconfiguration process.

Hence, up to this point reconfiguration has taken place only in the initiator subtree, while the operations in the other subtree have been carried out normally. In particular, new subscriptions may have been propagated in the second subtree, and some others might have been removed by unsubscriptions. This information has not reached the initiator subtree, since it was isolated. The views in the two subtrees must now be reconciled.

When RECMSG reaches the first end-point of the new link ($C$ in Figure 3), and after such dispatcher has updated its subscription table as described earlier, the first end-point activates the new link and sends to the second end-point a subscription for each event pattern in its subscription table (that we will call $P_{sub}$), followed by RECMSG, which still contains $P_{add}$.

At this point, the second end-point ($D$ in Figure 3) holds the whole set of information needed to perform the last part of the reconfiguration. In particular, it can determine what are the subscriptions that 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 from the set $P_{add}$ found in RECMSG all the subscriptions that are also in the subscription table of the second end-point of the new link. Unsubscriptions to these event patterns will have to be propagated back to the first subtree.

Moreover, it can determine what subscriptions are missing, because they have been generated meanwhile in the second subtree. Again, this is done by comparing the event patterns in $P_{add}$ against the subscription table. Subscriptions for such events will have to be propagated back to the first subtree, as well.

The last step of the process takes care of completing the reconfiguration by propagating this information to the first subtree, and by reconfiguring the tail path.

**Completing the Reconfiguration** At this point, the second end-point of the new link holds:

- the set of event patterns $P_{sub}$ for which the first end-point has sent subscriptions, and that must be propagated in the second subtree;
- the set of event patterns for which unsubscriptions must be propagated back to the first subtree;

- the set of event patterns for which subscriptions must be propagated back to the first subtree.

The propagation of all these subscriptions and unsubscriptions is carried out using normal subscription and unsubscription messages. As a matter of fact, subscriptions and unsubscriptions propagating from the second subtree to the first *must* propagate to the whole subtree, since they are normal subscriptions and unsubscriptions that never got a chance to propagate before. Instead, the subscriptions generated using $P_{sub}$, i.e., those coming from the first subtree towards the second one, will propagate automatically by providing just the right amount of reconfiguration. In fact, if no unsubscription has been issued, the subscriptions on the rest of the subtree are already routing events in the right direction, and hence subscriptions will propagate only on the tail path. However, if an unsubscription has been issued meanwhile, then the propagating subscriptions will rightfully propagate as needed outside the rest of the tree.

The only tile missing to complete the reconfiguration puzzle 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 the one from $D$ to $G$ in Figure 3). These subscriptions cannot be removed before the end of the reconfiguration path ($B$ in Figure 3) is reached, 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 dispatchers on the tail path. Hence, the second end-point of the new link, after having sent subscriptions and unsubscriptions back to the first subtree, and having propagated subscriptions in its own subtree, sends a RECMSG that will propagate along the tail path. When RECMSG is finally received by the last dispatcher on the reconfiguration path ($B$ in Figure 3), the latter behaves as if it had received an unsubscription message for each subscription in its table coming from the initiator, and processes it using the mechanisms used for normal unsubscription messages. This will eliminate superfluous subscriptions from the reconfiguration path, without the risk of removing necessary subscriptions, because the construction part of the reconfiguration process has already been completed.

**Avoiding Race Conditions on the Head Path** As previously mentioned, subscriptions are added on the head path in a non-standard way. A dispatcher can add the next dispatcher on the reconfiguration path as a subscriber without this latter dispatcher knowing it. This can cause problems if this latter dispatcher processes an unsubscription while RECMSG is being sent towards it. In fact the dispatcher could decide to unsubscribe from the sender of RECMSG before knowing that it should instead keep the subscription as a result of the reconstruction process. It is important to note that these unsubscriptions can be originated only by a subscriber in the subtree of the head path, since connectivity to the other subtree has not yet been restored.

Imagine that a dispatcher $D_2$ has previously sent a subscription to a neighbor $D_1$ for a given event pattern $p$, and that a reconfiguration process with the $D_1D_2$ link in the head path of the reconfiguration path is rebuilding subscriptions for pattern $p$. When $D_1$ receives RECMSG from its upstream neighbor in the reconfiguration path, it sees that it should add $D_2$ to its subscription table as a subscriber for pattern $p$. However, since $D_2$ is already a subscriber for pattern $p$, no action needs be taken and RECMSG is forwarded to $D_2$. In the meantime, while RECMSG is in transit, $D_2$ may decide to unsubscribe from the pattern $p$ and thus send the corresponding unsubscription message to $D_1$. This would cause $D_1$ to remove the subscription of $D_2$ from its subscription table, thus disrupting the reconfiguration process. This race condition arises because $D_2$ is not aware that the reconfiguration now demands that it should remain subscribed to $D_1$ to route events downstream towards the other subtree.

A viable solution is for dispatcher $D_1$ to ignore the unsubscription message coming from $D_2$, based on the knowledge that pattern $p$ is needed by the reconfiguration process. Unsubscriptions should be ignored by $D_1$ only until it is notified that $D_2$ has received and processed RECMSG. This solution can be realized by means of an *ignore table*, that contains the event patterns of the unsubscriptions that should be ignored. Moreover, an additional control message is needed to allow a dispatcher on the head path to notify its upstream neighbor that RECMSG has been processed, and hence that 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 that are already present due to a previous subscription. Thus, a dispatcher should include such 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, $D_1$ could receive a subscription message from $D_2$, immediately followed by an unsubscription, before RECMSG reaches $D_2$. In this case the subscription would cause no action, since it would have been already added by the reconfiguration, but the unsubscription would instead be processed, thus stealing the subscription added by the reconfiguration.

The solution is therefore to insert in the ignore table the patterns which are in $P_{add}$ and to which the recipient of RECMSG is already subscribed when such message is received, and subsequently add those patterns which are in $P_{add}$ and for which a subscription message is received while waiting for the control message. As soon as this is received, all the entries listed in the ignore table[4] should be removed from the ignore list.

A more conservative approach is to add to the ignore table all the patterns in $P_{add}$ before forwarding RECMSG,

[4]Actually, only the entries for a specific reconfiguration process should be removed, assuming that a dispatcher may be involved in more than one reconfiguration. An identifier associated to both RECMSG and the control message is sufficient. The details are included in the formalization in the Appendix.

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 becomes larger, but no special processing is needed when a subscription is received. This latter approach is the one used in the formalization in Appendix A.

## 5 Scenario-Dependent Implementation Details

Thus far, we have described the operation of our algorithm by glossing over the details related to how some functions, which are indeed necessary, can be provided. In particular, we did not specify how a link break can be detected or how a new route, that effectively re-establishes the connectivity of the tree, can be obtained. These details depend on the characteristics of the specific scenario where the algorithm is deployed, and are especially related to the nature of reconfiguration. In this section, we hint at some ways of providing this required functionality, in the context of the different scenarios described in Section 3.

### 5.1 Detecting a Lost Link

As mentioned in Section 3, a link between two nodes of the dispatching tree can disappear for essentially two reasons: either because it is being explicitly removed at the application level, or because a fault or a topological change in the underlying communication layer results in the inability to ensure communication between two nodes.

In the first case, detection of a missing link is not necessary since the end-points of the vanishing link are directly provided by the application and such information can be made readily available to our algorithm to trigger the subsequent reconfiguration phases. The second case may arise either in standard, wired networks or, even more frequently, in MANETs, where mobility naturally results in changes in the topology of the network. In this case, a number of solutions are reasonable to detect the link fault.

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 ad hoc networking (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[5] 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 that has to be monitored does not map directly to a single physical link. The difference is that in these cases a special point-to-point protocol

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

like ICMP can be used to implement the beaconing mechanism.

As an alternative to this proactive approach, that constantly monitors the network, a lazier approach can be exploited, that detects link breakages only when a communication failure is detected at the application level, e.g., by an exception returned while transmitting data on a socket. Obviously, this approach can be exploited only if the transport protocol adopted provides some form of reliability.

## 5.2 Choosing the Initiator

After a link has vanished from the tree, the problem becomes how to choose which of the end-points of the vanished link must start the reconfiguration process.

In the scenario where the reconfiguration is triggered by a decision taken at the application level the problem is usually solved trivially by simultaneously indicating which link is to be removed and which of its end-points is to initiate the reconfiguration process.

In the other cases, it is possible to take benefit of the fact that in every publish-subscribe middleware, dispatchers have an identifier used in subscription tables and each dispatcher knows the identifiers of its neighbors. Provided that an ordering function on identifiers can be determined, which is trivial since identifiers are often numbers, the most natural solution to the problem of determining the initiator is to choose the dispatcher that has the smaller (or greater) identifiers. Since each dispatcher knows the identifier of its neghbors, this decision can be taken locally and independently by each end-point.

## 5.3 Replacing a Missing Link With a New Route

After choosing the initiator, the problem becomes how to replace the removed link with another one that, by connecting the two separate subtrees which resulted from removing the link, re-establishes connectivity without creating loops.

Again, in the scenario where the reconfiguration is triggered by a decision taken at the application level the problem can be solved trivially. In fact, usually, as part of the decision of removing a link, the decision of which new link will substitute the removed one is also taken. Hence, there is no need to automatically discover a new route to join the two subtrees which result from removing a link, since the new link re-establishing connectivity is already specified at removal time. This also allows a reduction in the latency experienced by the user. Moreover, since link removal is essentially under the control of the application, special mechanisms can be put in place at the application level to ensure that the reconstruction could take place without additional link breaks. In this case, it would be conceivable to provide a single mechanism at the application level, by which the removal of a link and the insertion of a new link are effected as a single, atomic operation.

When the new route is not provided by the application level things become slightly more complicated. The initiator must request a new route to its neighbors; new routes have to be computed, possibly in a distributed way; they have to be delivered back to the initiator, which will choose one among them. A number of mechanisms can be used for this purpose, depending on the characteristics of the scenario.

If the publish-subscribe system is being used in a fixed network, it might be reasonable to suppose that each dispatcher maintains a cache of the network addresses of the dispatchers connected to its neighbors (i.e., each dispatcher has a local visibility of the system topology). When a link vanishes, the initiator can send around a special message to be propagated along the tree up to a certain number of hops. Such a message would include the list of the dispatchers which are known to be part of the disconnected subtree. Each dispatcher receiving such message could try to determine if it can reach one of the dispatchers that belong to the above mentioned list, and it would send back a reply including the network address of the "closest" dispatcher reachable, together with some information about the "distance" to that dispatcher. A protocol like ICMP could be exploited to determine such information. The initiator could collect such information and choose the best route to use. 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.

The same approach can be used also in presence of wireless links, if the required protocols are available, or it is possible to use the same strategy adopted by MAODV [13] for maintaining its multicast trees. This strategy makes heavy use of broadcast messages and does not require any special protocol except for the ability to send messages to immediate neighbors, which makes this strategy particularly suited for MANETs.

## 6 Discussion and Related Work

The algorithm described so far to reconfigure a tree of dispatchers in a publish-subscribe system is amenable to some enhancement and discussion of open issues to adapt it to more complex situations. In this section, we discuss such issues, describe how the same algorithm can be adapted to different settings such as multicast routing in ad hoc networks, and end with a description of related work.

## 6.1 Enhancements and Open Issues

Here, we discuss some enhancements that can be provided to our approach to adapt it to a wider range of scenarios or to improve its operation, together with some open issues.

### 6.1.1 Adding and Removing Dispatchers

The ability to manage link breaks and to allow an existing link to be replaced with a new one to adapt the dispatcher topology to changes in the external environments are crucial for long-running, large scale, publish-subscribe systems. On the other hand, another important issue to consider is how to manage the addition of new dispatchers and the removal of existing ones. This situation is trivial to manage if the dispatcher is to be added as a leaf of the tree or if a leaf is to be removed. The situation becomes much more complex if non-leaf dispatchers are to be added or removed.

First, we observe that typically a new dispatcher is added as the result of an explicit decision of the system manager. In such a situation, the dispatcher can be added at any arbitrary point in the tree by first inserting it as a leaf, then repeatedly applying the earlier presented algorithm to add and remove links to place the new dispatcher in the required position. A similar approach can be adopted to remove an existing dispatcher. Clearly, some optimized solutions are possible, that we are currently investigating

Another situation to consider is when a dispatcher suddenly becomes unreachable due to some unexpected event (e.g., a hardware fault of the host that runs the dispatcher). As a result of this situation the tree becomes partitioned in two or more pieces, depending on the number of neighbors the lost dispatcher had. The problem in this situation is how to choose the dispatchers to start the link reconstruction process and how to avoid cycles if more than one dispatcher starts this process in parallel. A possible solution to this problem comes by generalizing the approach taken by MAODV to address the loss of a link. The idea is to have a special dispatcher in the tree acting as the leader and to label each dispatcher with its logical distance from the leader (i.e., counting the number of hops between the leader and the dispatcher). The reconfiguration process is started by the neighbors of the vanished dispatcher farther from the leader (i.e., the ones that have a label greater than the label of the vanished dispatcher). Each of these dispatchers locates a route toward the leader subtree and uses our approach to reconfigure the subscriptions. In order to guarantee that no cycles are formed during the reconfiguration, the new link must have one end-point with a label smaller than that of the vanished dispatcher. This ensures that the link being added is to a node on the same tree as the leader, and not to a node on some other partition.

### 6.1.2 Managing Multiple Reconfigurations in Parallel

Thus far we have always considered the case where only one link is removed at a time. In practice, however, multiple simultaneous link breaks are possible. If the reconfiguration decision is taken at the application level, we can avoid the issue by disallowing multiple reconfigurations to overlap. In wired networks, the rate of link faults is low, and this issue is not likely to be a problem. However, in some environments, and particularly in MANETs, it may happen that several links break in parallel.

If the different reconfiguration processes do not overlap, the algorithm we proposed can be applied as is, without worrying about potential conflicts. The situation is much more complex when the different reconfiguration paths have one or more links in common or when an additional link break does not allow a running reconfiguration process to complete as expected. In the first case, we can exploit the special leader dispatcher described earlier to serialize multiple reconfigurations. Again, this is similar to MAODV's approach to merging multiple partitioned trees. Instead, when a link break occurs on a reconfiguration path, in addition to reconnecting the partitioned trees, we must ensure that the subscription paths also stabilize to a correct configuration. We are just beginning to explore these options by working with the formalization presented in the Appendix, analyzing the potential problems and searching for possible solutions. Meanwhile, we observe that because these situations are unlikely in several common settings (i.e., controlled reconfigurations and link breaks in wired networks) the algorithm proposed maintains its usefulness and it serves as a valid starting point for investigating the most critical scenarios.

### 6.1.3 Choosing Dispatchers

Up to this point we have restricted our discussion to the management of the dispatchers, but have ignored the issue of how to select which nodes act as dispatchers. In a fixed network, dispatchers can be chosen either directly by a system manager or placed on the network and connected to reflect the underlying topology. Typically the set of dispatchers is disjoint from the set of clients, making a clear separation between the processing required to distribute events and the systems exploiting the event distribution.

In the MANET environment, this distinction is not always possible, and it is likely that every node in the network will need to be able to act as a dispatcher in order to fully connect the subscribers as a tree of neighbors. In some cases, it is reasonable to force every subscriber to act as a dispatcher, while leaving open the option for including every publisher in the dispatching tree. In other cases, it may be possible to choose dispatchers which are not necessarily subscribers, but which have some special property making them good candidates for the dispatch task (e.g., longer battery life, faster computation). In this situation, the dispatchers may not be directly connected to one another in the underlying network, and may need to exploit some form of ad hoc routing to exchange send messages across multiple hops. Detecting link failures becomes more challenging in this case, but it may be require fewer overall reconfigurations because the underlying routing between dispatch nodes can hide some of the topology changes. What decision to take for selection of dispatchers is largely dependent on the characteristics of the application and the environment itself.

### 6.1.4 Extending the New Link

In the description of our reconfiguration approach in Section 4, the reconfiguration path is divided into three parts, namely head path, new link, and tail path, and the new link is defined as an ordered pair specifying the end-points of the link being inserted into the tree. This definition, while acceptable for the wired network where the routing infrastructure hides the details of communication between dispatchers, is not always applicable in the MANET environment. For example, assuming that the dispatching network maps directly to the network topology it is possible that one link will not be sufficient to connect the two partitioned parts of the tree, and additional intermediate nodes will need to be included to link the trees. In this case, the new link can be expanded beyond a single link and be described as a sequence of nodes where the head and tail of the sequence match the two end-points of the new link. Care must be taken during the propagation of the reconfiguration message along the new link to properly setup the subscription tables on the intermediate dispatchers to subscribe to all the events which must be exchanged between the two tree partitions.

## 6.2 Applicability

Our approach generalizes beyond reconfiguration in publish-subscribe systems to any kind of multipoint communication including standard multicast routing in the fixed network. Additionally, our ability to adapt to topology changes makes our approach appealing for multicast in MANETs. Consider MAODV [13], one of the most popular mobile ad hoc multicast routing algorithm. We observe that it maintains a separate multicast tree for each group. As a consequence of this choice, when a link breaks MAODV must reconfigure all affected trees separately, incurring a large overhead to discover new routes and perform the required reconfiguration steps independently for each group. In contrast, in our approach we build a single tree for all events. By applying this strategy in multicast routing and by using our algorithm to manage link failures it is possible to incur the overhead of finding a new route and reconfiguring the tree only once, even if this approach also requires updates to the dispatching information on top of the tree. By comparing the two approaches, we conjecture that in a network with many different multicast groups and significant overlap in the members of those groups the overhead of our approach could be less than that of an MAODV-like one. It is worth noting that the reverse mapping of applying multicast routing such as MAODV to publish-subscribe does not necessarily hold. This issue has been explored in detail within the context of fixed networks [9], and many of these observations carry also over into MANETs.

Another area to exploit our approach is in peer-to-peer applications [10]. One of the well-known problems in peer-to-peer applications is how to route information (e.g., queries and replies in file sharing applications, or messages in messaging applications) and how to reconfigure routes when new nodes are added or existing nodes are removed. Our approach comes into play here since most of the mechanisms we outline can be adapted to this setting. A further advantage of peer-to-peer networks not present in MANETs is that connection and disconnection of peers usually is kept under strict control of the application, users must press a button to join or leave the peer-to-peer network.

### 6.3 Related Work

The publish-subscribe approach itself has been gaining popularity in recent years, leading to the development of several middlewares. Most of these are targeted to local area networks and adopt a single, centralized dispatcher, or use multicast transport protocols to distribute events within the LAN. In recent years, the problem of wide-area event notification has attracted the attention of researchers [16] and some products have been presented which adopt a distributed dispatcher, such as TIBCO's TIB/Rendezvous, Jedi [5], Siena [4], READY [6], 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. The only exceptions are Siena [4] and the system described in [18], which adopt the trivial protocol of Section 4.3 to allow subtrees to be merged or trees to be split. Jedi [5] allows a different form of reconfiguration, namely allowing clients to be added, removed, or moved from host to host at run-time. Finally, even if current Elvin implementation (4.0.3) does not support any form of reconfiguration, some work [15] has been done to support mobile clients as in Jedi. In particular, a "proxy" to the Elvin server is being developed to support persistency of events to mobile clients that periodically disconnects.

Some research projects, including IBM Gryphon [1] and Microsoft Herald [3] have among their goals the development of a mechanism to obtain the same results we focused on in the work, but we were unable to find any public documentation illustrating existing results in the area.

## 7 Conclusions

Currently available publish-subscribe systems that adopt 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 external environment. In this paper we first provided an analysis of the possible sources of reconfiguration, then we presented a novel approach to solve the above limitation. The main advantage of our approach with respect to the trivial solution adopted by some publish-subscribe middleware is in greatly reducing the impact of reconfiguration on the event dispatching service. This is obtained by identifying the minimal set of dispatchers that have to be involved in the reconfiguration.

In this work we also examined several possible enhancements, together with some open issues that will be the subject of further research in the short-term. For instance, we are going to fully formalize our approach, not only to verify formally its correctness, but also to gain a better understanding about the delivery guarantees that it provides, and hence ideally drive the extensions of our base algorithm. Finally, we plan to investigate in more depth the application of our approach in the context of MANETs, and especially whether our approach can be successfully tailored to the problem of providing multicast routing in such environment.

## References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. 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. Johnson, and D. 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. 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. 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] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems—Middleware Workshop*, 1999.

[7] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, 1992.

[8] M.Corson, J.Macker, and G.Cinciarone. Internet-Based Mobile Ad Hoc Networking. *Internet Computing*, 3(4), 1999.

[9] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proc. of Middleware 2000*, 2000.

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

[11] C. Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet Draft, October 1999.

[12] 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 ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE97)*, number 1301 in LNCS, Zurich (Switzerland), September 1997. Springer.

[13] 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.

[14] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. 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*, Brisbane, Australia, May 2001.

[16] University of California, Irvine. *WISEN, Workshop on Internet Scale Event Notification*, Irvine, California, 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 International WWW Conference*, 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.

## A Formalization

This appendix contains the formalization of the algorithm described in Section 4. Figures 4 and 6 define several useful definitions, including both fundamental types and functions as well as more complex data structures needed for event and reconfiguration processing.

In the formalization, the algorithm itself is expressed as a set of actions, outlined in Figure 5. Three of these actions can be triggered externally by the system-dependent mechanisms described in Section 5, while the fourth action generalizes the receipt of a message. The details of each of these actions are provided in the final three figures with some useful macro definitions appearing in Figure 7.

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 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 actions of Figure 8 describe the operations common to all publish-subscribe systems in order to process subscriptions, unsubscriptions, and events; however, the definitions appearing here are augmented with processing specific to our reconfiguration process. In particular, the unsubscribe action references the ignore list.

Figure 9 shows the processing of the externally visible actions to add and remove a link from the dispatching tree, and Figure 10 contains the action responsible for starting a reconfiguration as well as the instructions for processing reconfiguration and control messages. While the actions of Figure 9 are present in some publish-subscribe systems, the actions of Figure 10 are unique to our approach.

**Sets**

| | |
|---|---|
| $P$ | 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 union of the two previous sets |
| $ID$ | the set of possible reconfiguration IDs |

**Symbol domains**

Unless otherwise specified the following symbols are defined as belonging to the following sets

| | | |
|---|---|---|
| A dispatcher | $d, d', ...$ | $\in D$ |
| A node | $n, n', ...$ | $\in N$ |
| A reconfiguration ID | $rId$ | $\in ID$ |

In each rule, $\mathsf{self} \in D$ denotes the current host, i.e. the one which is applying the rule

**Reconfiguration Path**

| | |
|---|---|
| $RP$ | denotes the 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:

| | |
|---|---|
| $\textsc{sub}(p)$ | a subscription message. |
| $\textsc{unsub}(p)$ | an unsubscription message. |

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

| | |
|---|---|
| $\textsc{recmsg}(rId, P_{add}, P_{del}, RP)$ | the reconfiguration message. |
| $\textsc{ctrlmsg}(rId)$ | the control 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 have to be deleted from the head of the reconfiguration path |
| $RP$ | the reconfiguration path |

**Functions**

| | |
|---|---|
| $\mathsf{newId}()$ | returns a fresh identifier |
| $\mathsf{send}(n, \mathsf{msg})$ | sends a message $\mathsf{msg}$ to a client or dispatcher $n$ |

The following functions applied to a sequence of dispatchers $\mathsf{Seq}$ return respectively

| | |
|---|---|
| $\mathsf{first}(\mathsf{Seq})$ | the first dispatcher in the sequence |
| $\mathsf{last}(\mathsf{Seq})$ | the last dispatcher in the sequence |
| $\mathsf{next}(\mathsf{Seq})$ | the dispatcher following $\mathsf{self}$ in the sequence |
| $\mathsf{prev}(\mathsf{Seq})$ | the dispatcher preceding $\mathsf{self}$ in the sequence |

**Figure 4. Definitions used in the formalization**

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

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

**Figure 5. Conditions for the activation of the actions**

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. |

**Figure 6. Tables maintained by each dispatcher**

```
//Process an unsubscription from node n
processUnsubFrom$(n, p) \equiv$
  $S \leftarrow S - \{(n, p)\}$
  if $\neg\exists n'(n' \neq n \wedge (n', p) \in S)$ then
    $\forall d | d \neq n($send$(d, \textsc{unsub}(p)))$
  else if $\exists!\ n'(n' \neq n \wedge (n', p) \in S \wedge n' \in D)$ then
    send$(n', \textsc{unsub}(p))$
```

```
//Send all the subscriptions in the subscription table to a new neighbor
sendSubsToNewLink$(d) \equiv$

  $P_{Add} \leftarrow \{p : \exists n(n \neq d \wedge (n, p) \in S)\}$
  $\forall p \in P_{Add}($send$(d, \textsc{sub}(p)))$
```

```
//Update the subscription table during a reconfiguration in the head path
updateSubTable$(P_{add}, d_{next}, P_{del}, d_{prev}) \equiv$

  $S \leftarrow S \cup \{(d_{next}, p), \forall p \in P_{add}\}$
  $S \leftarrow S - \{(d_{prev}, p), \forall p \in P_{del}\}$
```

**Figure 7. Auxiliary macro definitions**

```
//A subscription is received from a neighboring node n
subscriptionReceived$(n, \textsc{sub}(p))$
Action
  if $(n, p) \notin S$ then
    //n is not subscribed to p
    $S \leftarrow S \cup \{(n, p)\}$
    if $\neg\exists n'(n' \neq n \wedge (n', p) \in S)$ then
      //This is the first subscription received by self for pattern p
      $\forall d | d \neq n($send$(d, \textsc{sub}(p)))$
    else if $\exists! n'(n' \neq n \wedge (n', p) \in S \wedge n' \in D)$ then
      //This is the second subscription received by self for pattern p
      send$(n', \textsc{sub}(p))$
  else
    skip$(\textsc{sub}(p))$
```

```
//An unsubscription is received from a neighboring node n
unsubscriptionReceived$(n, \textsc{unsub}(p))$
Action
  if $(n, p) \in S \wedge \neg\exists rId((n, p, rId) \in I)$ then
    //n is subscribed to p and is not in the ignore table for pattern p
    processUnsubFrom$(n, p)$
  else
    skip$(\textsc{unsub}(p))$
```

```
//An event message is received from either a neighboring dispatcher or a local client n
eventReceived$(n, \textsc{event})$
Action
  $\forall p \in P |$match$(\textsc{event}, p)(\forall n' | n' \neq n \wedge (n, p) \in S($send$(n', \textsc{event})))$
```

**Figure 8. Actions for event dispatching and basic subscription management**

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

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

**Figure 9. Actions for handling link insertion and removal**

```
$//A\ link\ to\ dispatcher\ d\ is\ replaced\ by\ another$
substituteLinkTo$(d, RP)$
Action
  if self $\neq$ first$(newLink)$ then
    $//$self $does\ not\ coincide\ with\ the\ first\ end\text{-}point\ of\ the\ new\ link$
    $P_{add} \leftarrow \{p : (d,p) \in S\}$ $//\ the\ set\ of\ patterns\ to\ which\ d\ was\ subscribed\ along\ the\ old\ link$
    $rId \leftarrow$ newId()
    $I \leftarrow I \cup \{(\mathsf{next}(RP), p, rId), \forall p \in P_{add}\}$ $//\ add\ the\ patterns\ to\ the\ unsubscription\text{-}ignore\text{-}list$
    updateSubTable$(P_{add}, \mathsf{next}(RP), P_{add}, d)$
    $D \leftarrow D - \{d\}$

  else
    $//$self $is\ also\ the\ first\ end\text{-}point\ of\ the\ new\ link$
    $P_{add} \leftarrow \{p : (d,p) \in S\}$ $//\ the\ set\ of\ patterns\ to\ which\ d\ was\ subscribed\ along\ the\ old\ link$
    $rId \leftarrow$ newId()
    updateSubTable$(P_{add}, \mathsf{next}(RP), P_{add}, d)$
    $D \leftarrow D - \{d\}$
    $D \leftarrow D \cup \{\mathsf{next}(RP)\}$
    sendSubsToNewLink$(\mathsf{next}(RP))$

  $//Do\ not\ remove\ subscriptions\ needed\ by\ clients\ or\ other\ neighboring\ dispatchers$
  $P_{del} \leftarrow P_{add} - \{p : \exists n(n \neq d \wedge n \neq \mathsf{next}(RP) \wedge (n,p) \in S)\}$
  send$(\mathsf{next}(RP), \textsc{recmsg}(rId, P_{add}, P_{del}, RP))$

$//A\ dispatcher\ receives\ a\ reconstruction\ message\ from\ a\ neighboring\ dispatcher\ d$
recmsgReceived$(d, \textsc{recmsg}(rId, P_{add}, P_{del}, RP))$
Action
  if self $\in head$ then
    $//$self $is\ in\ the\ head,\ but\ will\ not\ be\ the$ first$(RP)$
    $I \leftarrow I \cup \{(\mathsf{next}(RP), p, rId), \forall p \in P_{add}\}$
    updateSubTable$(P_{add}, \mathsf{next}(RP), P_{del}, d)$
    send$(d, \textsc{ctrlmsg}(rId))$
    $P'_{del} \leftarrow P_{del} - \{p : \exists d'(d' \neq \mathsf{next}(RP) \wedge (d',p) \in S)\}$
    send$(\mathsf{next}(RP), \textsc{recmsg}(rId, P_{add}, P'_{del}, RP))$

  else if self $=$ first$(newLink)$ then
    $D \leftarrow D \cup \{\mathsf{next}(RP)\}$
    updateSubTable$(P_{add}, \mathsf{next}(RP), P_{del}, d)$
    send$(d, \textsc{ctrlmsg}(rId))$
    sendSubsToNewLink$(\mathsf{next}(RP))$
    $P'_{del} \leftarrow P_{del} - \{p : \exists d'(d' \neq \mathsf{next}(RP) \wedge (d',p) \in S)\}$
    send$(\mathsf{next}(RP), \textsc{recmsg}(rId, P_{add}, P'_{del}, RP))$

  else if self $=$ last$(newLink)$ then
    $D \leftarrow D \cup \{d\}$
    send$(\mathsf{next}(RP), \textsc{recmsg}(rId, \emptyset, \emptyset, RP))$
    $//Propagate\ the\ unsubscriptions\ received\ before\ tree\ was\ re\text{-}connected$
    $\forall p \in P_{add}(\textbf{if}\ \neg\exists n(n \neq d \wedge (n,p) \in S)\ \textbf{then}\ send(d, \textsc{unsub}(p)))$
    $//Propagate\ the\ subscriptions\ received\ before\ tree\ was\ re\text{-}connected$
    $\forall p(\textbf{if}\ \exists n((n,p) \in S) \wedge p \notin P_{add})\ \textbf{then}\ send(d, \textsc{sub}(p))$

  else if self $\in tail$ then
    send$(\mathsf{next}(RP), \textsc{recmsg}(rId, \emptyset, \emptyset, RP))$

  else if self $=$ last$(RP)$ then
    $D \leftarrow D - \{\mathsf{first}(RP)\}$
    $P_{Old} \leftarrow \{p : (\mathsf{first}(RP), p) \in S\}$
    $\forall p \in P_{Old}$(processUnsubFrom$(\mathsf{first}(RP), p)$)

$//A\ control\ message\ is\ received\ by\ a\ neighboring\ dispatcher$
ctrlmsgReceived$(d, \textsc{ctrlmsg}(rId))$
Action
  $I \leftarrow I - \{(d,p,rId), \forall d, p : (d,p,rId) \in I\}$ $//Remove\ d\ from\ the\ ignore\ table$
```

**Figure 10. Actions for dealing with link substitution**