Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration

Gian Pietro Picco
Politecnico di Milano, Italy
picco@elet.polimi.it

Gianpaolo Cugola
Politecnico di Milano, Italy
cugola@elet.polimi.it

Amy L. Murphy
Univ. of Rochester, NY, USA
murphy@cs.rochester.edu

Abstract

Distributed content-based publish-subscribe middleware provides the decoupling, flexibility, expressiveness, and scalability required by highly dynamic distributed applications, e.g., mobile ones. Nevertheless, the available systems exploiting a distributed event dispatcher are unable to rearrange dynamically their behavior to adapt to changes in the topology of the dispatching infrastructure.

In this work, we first define a strawman solution based on ideas proposed (but never precisely characterized) in existing work. We then analyze this solution and achieve a deeper understanding of how the event dispatching information is reconfigured. Based on this analysis, we modify the strawman approach to reduce its overhead. Simulations show that the reduction is significant (up to 50%), and yet the algorithm is resilient to concurrent reconfigurations.

1 Introduction

Publish-subscribe middleware is gaining popularity because the asynchronous, implicit, multi-point, and peer-to-peer communication style it fosters is well-suited for modern distributed computing applications. While the majority of deployed systems is centralized, commercial and academic efforts are focusing 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 systems. 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 is true in practice only if the publish-subscribe *system* is itself able 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 behavior accordingly.

The vast majority of currently available publish-subscribe middleware has ignored this problem thus far. The only exception is a very simple approach (henceforth referred to as the *"strawman approach"*) suggested by some authors [3, 17] but whose details have not been defined or implemented to date. Hence, in this paper we put forth our contributions by:

- providing a precise characterization of the strawman approach, and elucidating the principles underlying reconfiguration;
- presenting a novel solution that builds upon the straw man approach, but modifies its processing to obtain a significant reduction in overhead while still tolerating frequent reconfigurations;
- validating our solution through simulation, by comparing it against the strawman approach.

The paper is structured as follows. Section 2 contains a concise introduction to publish-subscribe, a discussion about the possible sources of reconfiguration, and a description of related efforts in the field. Section 3 defines the reconfiguration problem we tackle in this paper. Section 4 describes the aforementioned strawman solution, which is then analyzed in Section 5, leading to the optimizations we propose and illustrate in the same section. Section 6 presents the simulation results validating our approach. The paper ends with the brief concluding remarks in Section 7.

2 Background and Motivation

In this section we provide an overview of publish-subscribe systems, describe the reconfiguration scenarios that motivate our work, and survey related 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. Recently, many publish-subscribe middleware have become available, which differ along several dimensions[1]. Two are usually considered fundamental: the expressiveness of the subscription language and the architecture of the event dispatcher.

The expressiveness of the subscription language draws a line between *subject-based* systems, where subscriptions identify only classes of events belonging to a given channel or subject, and *content-based* systems, where subscriptions contain expressions (called *event patterns*) that allow sophisticated matching on the event content. Our approach 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.

The architecture of the event dispatcher can be either centralized or distributed; in this paper, we focus on the latter case. In such middleware (see Figure 1) a set of *dispatching servers*[2] are interconnected in an overlay network and cooperatively route subscription and event messages.

The systems exploiting a distributed dispatcher can be further classified according to the interconnection topology of dispatchers and the strategy exploited for message routing. In this work we consider a subscription forwarding scheme on an unrooted tree topology, as this choice covers the majority of existing systems.

In a subscription forwarding scheme [3], subscriptions are delivered to every dispatcher, along a single unrooted tree spanning all dispatchers, and are used to establish the routes that are followed by published events. When a client issues a subscription, a message containing the corresponding event pattern is sent to the dispatcher the client is attached to. There, the event pattern is inserted in a subscription table together with the identifier of the subscriber, and the subscription is forwarded to all the neighbors. During this propagation, the dispatcher behaves as a subscriber with respect to the rest of the dispatching tree. Each dispatcher, in turn, records the event pattern and re-forwards the subscription to its neighbors, except for the one that sent it. This scheme is usually optimized by avoiding subscription forwarding of the same event pattern in the same direction[3].

[1]For more detailed comparisons see [3, 5, 10].

[2]Hereafter we refer to *dispatching servers* simply as *dispatchers*, although the latter refers more precisely to the whole distributed component in charge of dispatching events, rather than to a specific server part of it.

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

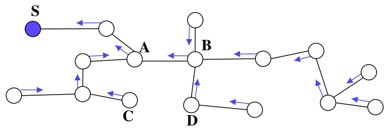


Figure 1. Subscription forwarding.

This process effectively sets up a route for events, through the reverse path from the publisher to the subscriber. Requests to unsubscribe from a given event pattern are handled and propagated analogously to subscriptions, although at each hop entries in the subscription table are removed rather than inserted.

Figure 1 shows a dispatching tree where a dispatcher (the dark one) is subscribed to a certain event pattern. The arrows represent the routes laid down according to this subscription, and reflect the content of the subscription tables of each dispatcher. To avoid cluttering the figure, subscriptions are shown only for a single event pattern. Here and in the rest of the paper, we ignore the presence of clients and focus on dispatchers, not only to simplify the treatment, but also because in the domains we target there is often no distinction between the two. Accordingly, even if in principle only clients can be subscribers, with some stretch of terminology we say that a dispatcher is a subscriber if at least one of its clients is a subscriber. Moreover, we assume that the links connecting the dispatchers are FIFO and transport reliably subscriptions, unsubscriptions, events, and other control messages. Both assumptions are typical of mainstream publish-subscribe systems, and are easily satisfied by using TCP for communication between dispatchers.

2.2 Sources of Dynamic Reconfiguration

Publish-subscribe systems are intrinsically characterized by a high degree of reconfiguration, determined by their very operation. 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, and their effect is the vanishing and/or appearance of one or more links between dispatchers. The simplest scenario is where reconfiguration is triggered at the application layer. For instance, the publish-subscribe systems deployed in enterprise usually rely on a backbone of interconnected dispatchers. A system administrator may need to substitute 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 effect should be an automatic reconfiguration of the distributed dispatcher to adapt event routes to the new topology.

Unfortunately, the source of reconfiguration is not always under the control of applications. A dispatcher may become disconnected from one of its neighbors because the physical link connecting the two has failed. Mobile computing defines a scenario where this is particularly likely to happen, since the network topology can change dynamically as mobile hosts move and yet remain connected through wireless links. This is brought to an extreme by mobile ad hoc networks (MANETs), where the networking infrastructure is totally absent and connectivity is determined solely by the distance between hosts. In all these cases, lack of communication with a dispatcher results in the inability to route messages towards it, due to partitioning of the dispatching tree. The reconfiguration process must restore the connectivity of the overlay network, but also properly rearrange the routing information contained within it.

A somehow intermediate scenario is provided by peer-to-peer systems, where the ability to perform scalable content-based event routing can be exploited to implement data sharing applications. This idea has been exploited by some of the authors in the PeerWare [6] middleware, and is also described by other researchers in [8]. In this setting, each peer routes messages much like a dispatcher. Consequently, the underlying routing mechanism must be able to cope with frequent changes of the topology of the peer overlay network, as users join and leave the system.

2.3 Related Work

Most publish-subscribe middleware are targeted to local area networks and adopt a centralized dispatcher. In recent years, the problem of wide-area event notification attracted the attention of researchers [14] and systems exploiting a distributed dispatcher became available, such as TIBCO's TIB/Rendezvous, Jedi [5], Siena [3], READY [7], Keryx [16], Gryphon [1], and Elvin4 [12]. To the best of our knowledge, none of them provides any special mechanism to support the reconfiguration addressed by this paper.

The closest match is the work on Siena [3] and the system described in [17]. These papers briefly suggest the use of the strawman solution to allow subtrees of dispatchers to be merged or trees to be split, but they do not provide details about its design, let apart providing an implementation or a validation through simulation. In this paper, we elaborate and provide details about the ideas presented in the aforementioned papers, and use the strawman solution as a term of comparison for our own solution. Jedi [5] provides a different form of reconfiguration that allows only clients (not dispatchers) to be added, removed, or moved to a different dispatcher at run-time. Similarly, Elvin [13] supports mobile clients through a proxy server, although this feature is not included in the latest (4.0.3) release. Finally, some research projects, such as IBM Gryphon [1] and Microsoft Herald [2], claim to support a notion of reconfiguration similar to the one we address in this work, but we were unable to find any public documentation about existing results.

3 The Reconfiguration Problem

We view the problem of dynamically reconfiguring a publish-subscribe system as composed of three subproblems that involve the:

1. reconfiguration of the dispatching tree, to retain connectivity among dispatchers without creating loops;

2. reconfiguration of the subscription information held by each dispatcher, to keep it consistent with the changes in the tree without interfering with the normal processing of subscriptions and unsubscriptions;

3. minimization of event loss during reconfiguration.

The objective of this paper is to provide an efficient solution for correctly reconfiguring the subscription information, i.e., for the second of the aforementioned problems. The rationale for this choice lies in the fact that *maintaining the consistency of subscription information is the defining problem of content-based publish-subscribe systems*: if the information necessary for event dispatching is misconfigured, or propagated inefficiently, the whole purpose of a content-based system may be undermined.

To understand why, it is useful to compare how subject-based and content-based systems are typically implemented. In subject-based systems, a spanning tree is built for each subject, connecting all the dispatchers subscribed to it. Once this tree is built, dispatching is trivial: an event matching a subject is simply broadcast over the corresponding tree. Notably, this closely resembles multicast systems, which indeed provide a suitable technology for implementing subject-based publish-subscribe systems. In content-based systems, instead, the number of "subjects" (i.e., the event patterns) is not known a priori and is potentially infinite, since it is entirely determined by the clients. Moreover, an event may match several event patterns at once. For these reasons, it becomes unreasonable to build a spanning tree for each event pattern [9]. A better solution, exploited by most content-based systems, is to build a single tree connecting all the dispatchers, and decorate such a tree with the information (i.e., the subscriptions) necessary to route events over it, as we described in Section 2.1.

Hence, as mentioned earlier, the availability of a tree connecting all the dispatchers is a precondition for content-based systems, but it is not the distinctive feature. In the remainder of this paper, we assume that the underlying tree is kept connected and loop-free by some other algorithm, and concentrate on the event dispatching issue. This is further supported by the observation that algorithms to maintain connectivity of the underlying tree strongly depend on the specific reconfiguration scenario (e.g., MANET vs. peer-to-peer computing over the Internet), and therefore decoupling the event dispatching issue from the tree reconfiguration issue is actually beneficial.

This, however, does not mean that we are disregarding this tree maintenance problem. Since our ultimate goal is the development of a publish-subscribe system for highly dynamic environments, like those defined by mobile computing and peer-to-peer networks, our ongoing activities are investigating how existing algorithms (e.g., those developed for MANET routing [11] or IP multicast) can be adapted to our needs. Moreover, we are developing a layer on top of our solution that is based on epidemic algorithms [4], and is in charge of recovering events lost during reconfiguration.

Next we describe the aforementioned strawman solution which, albeit currently not implemented by any system, is the only point of comparison available in the literature.

4 A Strawman Approach

The idea of the strawman approach [3, 17] is to carry out the reconfiguration by using exclusively the primitives available in a publish-subscribe system. In particular, the reconfiguration triggered by a link removal can be dealt with 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. Note how the only information needed regarding the overlay network being reconfigured is the notification that a link towards a dispatcher has appeared or vanished.

This strawman approach is the most natural and convenient when reconfiguration involves only either an isolated link insertion or removal, thus leading to a merging of two dispatching trees or a partitioning of one. Nevertheless, it exhibits a number of drawbacks when applied to arbitrary reconfigurations, e.g., those where multiple link breaks and insertions occur randomly and concurrently, or those where a broken link is shortly replaced (through the underlying tree reconfiguration mechanism) by a new link established in a different portion of the tree.

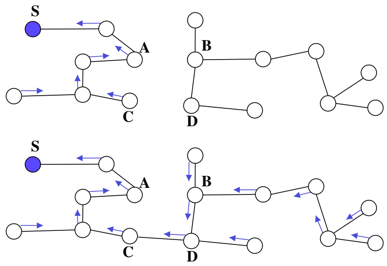


Figure 2. The dispatching tree of Figure 1 during and after a reconfiguration performed using the strawman approach.

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 approach, when the link (A, B) is removed the two end-points trigger unsubscriptions in their subtrees, without taking into account the fact that a new link will be found between C and D, effectively reconnecting the two subtrees. 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 after most of these subscriptions will be rebuilt by the insertion of the new link (C, D). The resulting reconfiguration of subscription information is not only inefficient, but it may greatly increase the loss of events.

In the next two sections, we show how simple modifications to the strawman approach, driven by an in-depth analysis of its operation, lead to significant improvements in the process of reconfiguring event dispatching information.

5 A More Efficient Approach

To design a new algorithm for highly dynamic environments, we begin by observing that the drawbacks of the strawman algorithm described in Section 4 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. 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 coordination could improve the performance of the strawman approach without sacrificing consistency when multiple link breaks occur in parallel.

5.1 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[4]. 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 4: *a subscription issued by a client is propagated in the dispatching tree following the unique route 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.

To understand this rule we observe that, for each event pattern p, there exists a minimum spanning tree containing all the dispatchers subscribed to p. For instance, in Figure 3 this minimum spanning tree is made 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 entry point for a new subscriber to such minimum 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.

These rules prompt two observations. First, the price of 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 observations lead to a criteria for designing a reconfiguration algorithm: keeping the tree "dense" of subscriptions, thus reducing the overhead caused by the propagation of subscriptions.

[4]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 assume the usual optimization of avoiding to forward a subscription already present in the system.

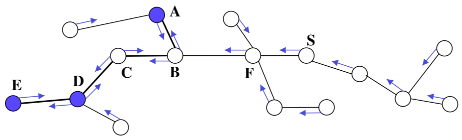


Figure 3. How new subscriptions propagate on the dispatching tree.

Our solution embodies this criteria by leveraging off of the conventional subscription and unsubscription operations like the strawman one, but by performing them in the inverse order: the subscriptions triggered by the appearance of a link are issued immediately, while the unsubscriptions due to a link break are issued only after a given delay. In our solution, the delay is provided by a timeout, which is initialized on both involved dispatchers when a link breaks, and whose expiration triggers the propagation of unsubscriptions. Ideally, this delay should coincide with the time needed by the underlying tree maintenance algorithm to restore the connectivity of the tree, by somehow replacing the broken link with the insertion of another one.

This way, subscriptions persist longer in the tree during a reconfiguration. It is true that this strategy may add subscriptions that must be removed immediately after, but in any case these subscriptions propagate only up to the first splitter. Moreover, this solution has the beneficial side effect of reducing the disruption of event routes, since it is unlikely that a subscription is removed only to be restored shortly after. Nevertheless, some additional care must be taken for reconfigurations where broken links and new links have a dispatcher in common, as explained in the following.

5.2 About Links Sharing a Dispatcher

The considerations presented in the previous section enable relevant advantages over the strawman approach. However, the fact that subscriptions are now always processed before unsubscriptions may be very inefficient in the case where one end-point of one or more broken links coincides with one end-point of one or more new links.

To understand why, let us consider the scenario in Figure 4. If subscriptions are processed before unsubscriptions when the link (A, B) vanishes and is replaced by (A, D), then A and D should exchange their subscriptions across the new link. However, since unsubscriptions have not yet been processed, the only subscription of A is the one routing events towards the vanished link. If propagated, this subscription would cause the creation of several "wrong" subscriptions (the dashed arrows in Figure 4) routing events

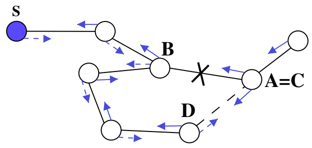


Figure 4. A case where one end-point of the broken link and one of the new link coincide.

from B's semitree towards A. These subscriptions are useless and will be removed by the unsubscription process, since there are no subscribers in A's semitree. The strawman solution does not suffer from the same problem since it processes unsubscriptions before subscriptions; A's subscription is removed first and therefore it is never propagated unnecessarily.

The scenario we depicted, which at first may seem a very special case, is actually quite common. If we disregard the dispatcher attached to the right of A in Figure 4, then A is a leaf dispatcher being detached and reattached to a different node. Leaf nodes are usually a large fraction of the total number of dispatcher and, since they are at the fringe of the system, they are more subject to reconfiguration.

Fortunately, this case can be dealt with fairly easily. Reconfiguration can be optimized by not forwarding subscriptions directed *only* towards dispatchers connected through links that are now broken. All the other subscriptions, i.e., those coming from clients attached to the shared dispatcher and those associated to intact links, are propagated as usual. An easy way to accomplish this processing, and yet delay the propagation of unsubscriptions, consists of immediately removing subscriptions as soon as a link breaks, and delay their propagation until the aforementioned timeout, set by dispatchers upon link breakage, expires. If the shared dispatcher is a leaf the processing is the same but no unsubscription need be propagated upon expiration of the timeout. In fact, the only link connecting the dispatcher to the rest of the tree is now broken, and there is no dispatcher to propagate unsubscriptions to.

Armed with this knowledge, we are now ready for a detailed description and analysis of our solution.

5.3 Rearranging Subscription Tables

As in the case of the strawman solution described in Section 4, the reconfiguration process takes place when the overlay network maintenance layer notifies dispatchers of the existence of a new or broken link. In the former case, our solution behaves exactly like the strawman solution, by propagating all subscriptions across the new link. Instead, when a broken link is detected, the subscriptions laid towards it are now immediately removed from the local subscription table, but not propagated. If the dispatcher at hand is a leaf, no further processing is required. Otherwise, a timeout associated to the broken link is set. As we discussed earlier, the timeout has the only effect of delaying the forwarding of unsubscriptions. Upon timeout expiration, our solution takes care only of forwarding unsubscriptions, since they have already been removed locally upon detection of the broken link.

Notably, the reconfiguration described by this algorithm does not interfere with the normal processing of events and (un)subscriptions. In fact, it relies 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.

As we discussed before, this algorithm makes sure that subscriptions rerouting events through the new link are laid down *before* the obsolete subscriptions that were needed only to route events through the vanished link are removed. Clearly, this latter unsubscription step cannot be delayed forever, and its occurrence is governed by the timeout value. This parameter plays a crucial role in our solution. If the value is too small, our solution approaches the strawman one because unsubscriptions are triggered too early, before subscriptions have been restored. If it is too large, obsolete routes are going to remain in place and steer events where there are no subscribers, thus increasing overhead.

In the next section, we analyze through simulation the effect of the timeout and several other parameters.

6 Simulation Results

We defined our simulations with two different goals in mind. The first goal was to *verify* that our algorithm behaves correctly in the presence of reconfiguration. The second goal was to *evaluate* the performance of our algorithm in terms of minimizing overhead, using the strawman solution described previously as a baseline for comparison. These two goals were satisfied by two sets of simulations, measuring the percentage of events correctly delivered, and the number of messages exchanged during reconfiguration.

6.1 Simulation Setting

Published work on publish-subscribe systems rarely presents validation of results through simulation. In the absence of reference scenarios for comparing these systems we defined our own, based on what we believe are reasonable assumptions covering a wide spectrum of applications.

Events, subscriptions, and matching. Events are represented as randomly-generated 9-character strings, where each character can be any of the (96) printable characters. Subscriptions are represented as a single character.

An event matches a subscription if it contains the character specified by the subscription. Hence, 96 different subscriptions are available in the system[5]. Nevertheless, each dispatcher can subscribe to at most s subscriptions drawn randomly from the 96 available. In our simulations, we chose s=10, since in a content-based system it is unlikely that a subscription is shared among several subscribers.

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

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

Tree topology. The results we present here are all obtained with tree configurations up to 200 dispatchers, each connected with at most four others. Simulation runs with different values showed that the influence of this parameter is negligible. The links connecting two dispatchers are assumed to behave as an error-free 10 Mbit/s Ethernet link.

Clients are not modeled explicitly, as their activity affects only the dispatcher they are attached to. Moreover, in the scenarios we target (e.g., MANET and peer-to-peer networks) the publish-subscribe system is likely to be deployed so that clients and dispatchers coincide.

Tree reconfiguration. In our simulations we are not interested in cases where a dispatching tree becomes partitioned or merged with another, as in this situation the behavior of the two approaches is equivalent. Instead, we evaluate their performance when a broken link is eventually replaced by another. The selection of the links breaking or appearing is done randomly. However, the same random sequence was applied in both approaches. To retain some degree of control about when a reconfiguration occurs, we assume that each broken link is replaced by a new one in 0.1 s. While this is not necessarily a good representation of reality, the bias introduced by this assumption affects both approaches in the same way, and hence does not influence our results. In any case, we verified that the algorithm works also for arbitrary link breakages and insertions.

Reconfigurations are allowed only in the interval between 3 and 7 seconds, with a frequency determined by the duration of the interval between two reconfigurations, ρ. We consider two scenarios: $\rho = 0.3s$, which yields non-overlapping reconfigurations, and $\rho = 0.03s$, where several reconfigurations may overlap. The latter can be regarded as an approximation of the case where a non-leaf dispatcher is detached from the tree and multiple links are broken at once. In any case, it defines a difficult reconfiguration scenario providing a good, extreme test case for our analysis.

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

Simulation tool. In this paper we compare the two approaches only at the application level, and we are not concerned about the underlying networking stack. Hence, we decided to develop our simulations using OMNET++ [15], a free, open source discrete event simulation tool.

6.2 Measuring Event Delivery

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

This is indeed the behavior resulting from our simulations, shown in Figure 5. The top plot is obtained in the scenario with $\rho = 0.3s$, i.e., non-overlapping reconfigurations. In this case, a correct event routing is re-established right after each reconfiguration. Instead, the bottom plot shows the system behavior when $\rho = 0.03s$. Despite the barrage of concurrent, overlapping reconfigurations, the system quickly returns to stable after the last reconfiguration.

The measurements were performed by relying on a subset of the dispatchers belonging to a *stable core*. Dispatchers in the stable core have a more constrained behavior than the others, since they are forbidden to (un)subscribe after a given time threshold σ, whose value is set to $2s$ in our tests. Measurements about event delivery are constructed

[5] We assume a single application deployed on the tree.

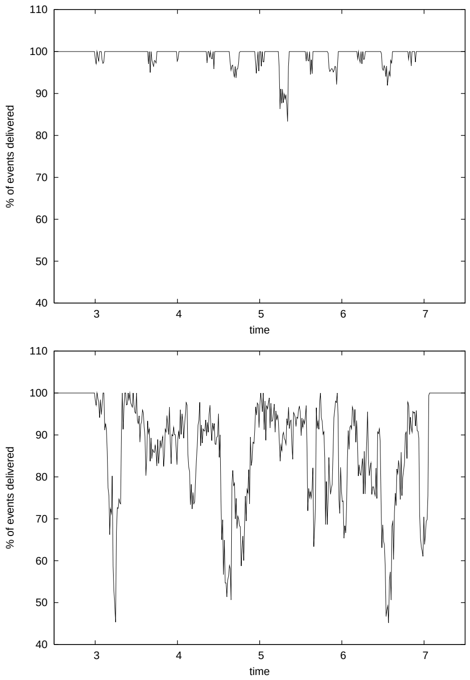


Figure 5. Event delivery during non-overlapping (top) and overlapping (bottom) reconfigurations.

by computing the ideal set of recipients for each published message, and comparing it with the actual number of copies received. The ideal set of recipients can be computed easily based on knowledge of the subscription tables of dispatchers in the stable core, since this information is required to remain stable after σ. Moreover, since only the stable core is subject to this limitation, the algorithm is validated not only against the reconfiguration coming from changes in the topology, but also for the reconfiguration of routing information determined by the (un)subscriptions coming from dispatchers not in the core. The results in Figure 5 are derived with 100 dispatchers, 50% of which belong to the core. Moreover, 50% of the dispatchers inside the core, and 50% of those outside the core are subscribers. The event load is assumed to be large, with $f_{pub} = 0.05$ (i.e., about 50 publish/s). Finally, the timeout is $T = 0.11s$.

6.3 Measuring Reconfiguration Overhead

The amount of reconfiguration overhead is the metric that we need to compare our optimized solution against the strawman one. The overhead is determined only by: *i)* the (un)subscription messages being exchanged because of reconfiguration; *ii)* the event messages being misrouted along obsolete subscriptions. In the following, we report about the improvement we achieve over the strawman solution, examine the aforementioned sources of overhead in more detail, and see how the timeout parameter T affects them.

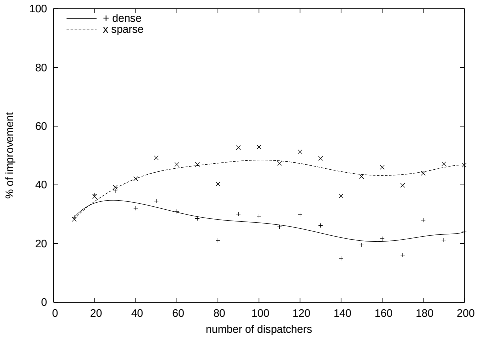
In the simulation results we are about to present, the overhead generated by messages is calculated in terms of the number of hops they traveled. Thus, for instance, a subscription issued by a dispatcher generates an overhead equal to the number of hops traveled by the subscription message. Moreover,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

Overall improvement. Figure 6(a) shows the percentage of improvement of our solution against the strawman one, in a configuration with $f_{pub} = 0.001$, $\rho = 0.3s$, and $T = 0.15s$. The original data points are reported together with their Bezier interpolation to help visualize the overall trend.

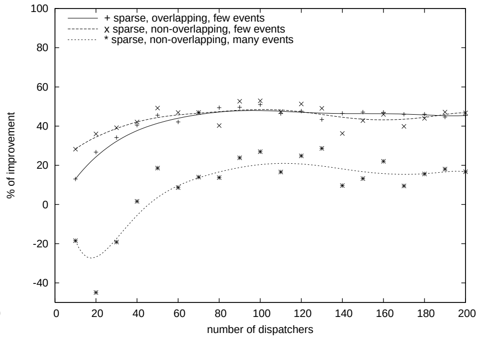
By showing the results for dense (80%) and sparse (20%) configurations of subscribers, the chart in Figure 6(a) confirms our intuition that the optimization we propose is even more efficient when subscribers are sparse. In fact, in this case the splitters are farther away, and the unnecessary (un)subscriptions propagated by the strawman solution travel to a greater extent. Moreover, there are also greater chances that a splitter does not exist—a case where the strawman solution performs particularly poorly.

To give a feel of the relevance of the percentage improvement, in a dense tree with 200 dispatchers there are about 3,500 messages (on the average, with a peak of 8,000) being exchanged during the reconfigurations, while in a sparse tree there are about 5,300 (with a peak of 18,000). Hence, even our lowest improvement of 20% may already lead to a significant reduction in the traffic overhead.

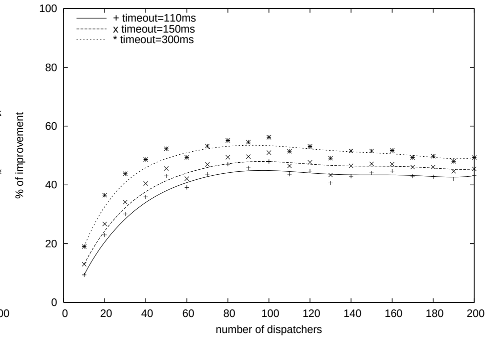
Figure 6(b) shows how the overhead is affected by the frequency of reconfiguration and the event load. The scenario considered is with sparse subscribers, and $T = 0.15s$. It is interesting to note how the improvement we obtain is essentially independent from the frequency of reconfiguration. Even with the barrage of overlapping reconfigurations induced by an interval $\rho = 0.03s$, our solution achieves basically the same percentage improvement as before, at least after 70 dispatchers. Before this limit, performance is worse essentially because, with few dispatchers, the tree is so disrupted that the optimizations introduced by our approach have a reduced impact. Nevertheless, in this case the actual overhead reduction, as opposed to the percentage improvement, is even greater, since the number of messages



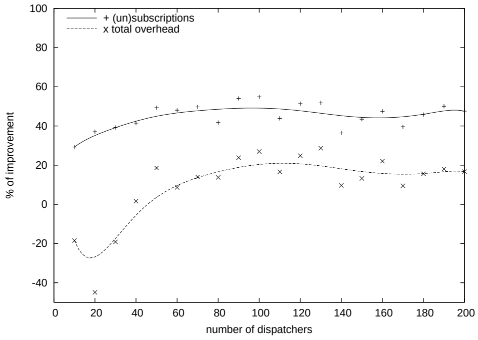
(a) % of improvement vs. density of subscribers.



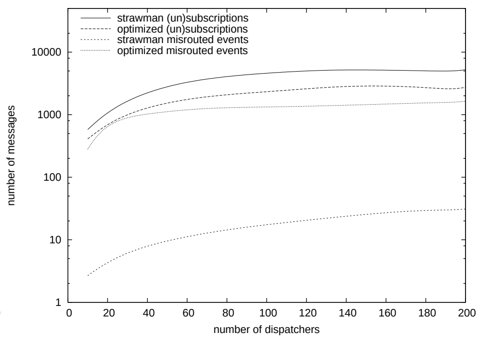
(b) % of improvement in various scenarios.



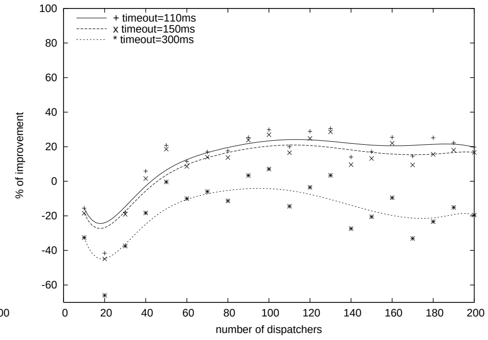
(c) Impact of timeout with low publish rate and overlapping reconfiguration.



(d) Impact of misrouted events on overhead, in terms of percentage improvement.



(e) Impact of misrouted events on overhead, in terms of messages exchanged.



(f) Impact of timeout with high publish rate and non-overlapping reconfigurations.

Figure 6. Measuring overhead and comparing it with the strawman solution.

caused by reconfiguration is much larger. For instance, in a configuration with 200 dispatchers, the strawman solution needs (on average) 63,162 messages to reconfigure, while our solution needs only 34,458. Instead, the percentage improvement obtained by our solution is lower when the event load is high, as in the case with $f_{pub} = 0.05$ shown in Figure 6(b). The reason for this behavior is that a high publish rate increases the overhead caused by misrouted events. On the other hand, it should be observed that a high publish rate results in a huge event load, to the point of making irrelevant any kind of optimization regarding the reconfiguration of dispatching information. For instance, in the case with many events shown in Figure 6(b), the total number of events flying in the system during the reconfiguration period is 2,025,756 (368,319 events/s are being dispatched in the system), while the total overhead generated by the strawman solution is only 11,028 messages.

(Un)Subscriptions. The hypothesis underlying our algorithm is that the number of (un)subscription exchanged during reconfiguration is significantly lower than that needed by the strawman solution. This hypothesis is actually verified by our simulations, e.g., as shown by Figure 6(a), where the impact of misrouted events is negligible. However, the effect of the timeout parameter over (un)subscriptions remains to be seen. The answer to this question is provided by Figure 6(c). This chart is obtained by assuming a sparse configuration of subscribers (20%), a low publish rate ($f_{pub} = 0.001$), and overlapping reconfigurations ($\rho = 0.03s$). It is easy to see how, in this scenario, increasing the timeout improves performance: about a 10% improvement is obtained by simply increasing the timeout from $T = 0.11s$ to $T = 0.3s$. The reason for this is interesting: under frequent reconfiguration, unsubscriptions from the end-points of the old link are triggered much later than in the strawman case. During this extra time, other concurrent reconfigurations may cause the appearance of new links, and the consequent propagations of subscriptions. Hence, when the unsubscriptions should be finally propagated, they are likely to be "short-circuited" by the existence of the new subscriptions. This largely reduces the traffic caused by subscriptions that are removed and subsequently restored by a concurrent reconfiguration. This phenomenon explains also why the effect of the timeout on non-overlapping reconfigurations is negligible—the reason for not showing specific charts of this situation.

Misrouted Events. As we mentioned earlier, when events are published at a high rate our solution achieves a lower improvement, because a lot of events get misrouted through obsolete routes in the time interval between a link break and the corresponding timeout expiration. Figure 6(d) shows graphically the impact of misrouted events, in a scenario with $f_{pub} = 0.05$, sparse subscribers, non-overlapping reconfigurations, and $T = 150ms$. The chart evidences how the significant reduction achieved for (un)subscriptions does not yield a comparable overall improvement, due to the presence of misrouted events. Figure 6(e) evidences the same phenomenon from a different angle, by comparing the actual number of messages generated by the two approaches, and shows how the number of misrouted events is actually comparable to the number of (un)subscriptions.

Nevertheless, the timeout parameter may play a significant role in reducing also the impact of misrouted events, as illustrated by Figure 6(f). The chart is obtained in the scenario defined by the same parameters used for Figure 6(d) and 6(e), but shows how the percentage improvement varies along with the timeout. The chart confirms the intuition that, with a high publish rate, the smaller the timeout the better, since it minimizes the time interval during which events are duplicated along obsolete routes. In the situation of Figure 6(f) the best performance is provided by a timeout $T = 110ms$, i.e., close to the time needed to restore the tree connectivity ($100ms$), while a timeout of $T = 300ms$ provides the worst performance.

7 Conclusions

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 topological changes. Solutions available in the literature at best mention a strawman solution whose simplicity is often outweighed by its inefficiency, since it involves areas that should not be affected by reconfiguration.

In this work, we started by characterizing precisely the strawman approach and analyzing its performance in different scenarios. We then described some modifications to the strawman approach and evaluated these optimizations by running a large set of simulations, whose results show that we are able to achieve an overhead reduction of up to 50%, while remaining resilient to multiple concurrent reconfigurations involving several nodes. Simulations also allowed us to characterize precisely the cases where the strawman solution could be preferred, thus contributing to clarify the impact of the different parameters on the overall system performance in the presence of reconfigurations.

Future work on this topic will complement the results presented in this paper with mechanisms to provide overlay network maintenance, and with solutions for recovering lost events. For the latter problem, we have already developed a solution [4] based on epidemic algorithms. All these efforts are finding their natural exploitation in the implementation of a new generation content-based publish-subscribe middleware supporting our approach to reconfiguration.

References

[1] G. Banavar et al. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proc. of ICDCS*, 1999.

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

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

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

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

[6] G. Cugola and G. Picco. Peerware: Core middleware support for peer-to-peer and mobile system. Technical report, 2002. Available at `www.elet.polimi.it/~picco`.

[7] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proc. of the Middleware Workshop at ICDCS*, 1999.

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

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

[10] D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of ESEC/FSE97*, LNCS 1301. Springer, 1997.

[11] E. Royer and C. Perkins. Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol. In *Proc. of MobiCom99*, 1999.

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

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

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

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

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

[17] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for Internet-scale event services. In *Proc. of the 8^{th} WETICE Workshop*, 1999.