Content-Based Routing for Publish-Subscribe on a Dynamic Topology: Concepts, Protocols, and Evaluation

Gianpaolo Cugola, Davide Frey, Amy L. Murphy, and Gian Pietro Picco

G. Cugola, D. Frey, and G.P. Picco are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, P.za L. da Vinci, 32, 20129 Milano, Italy. E-mail: {cugola,frey,picco}@elet.polimi.it. A.L. Murphy is with the Faculty of Informatics, University of Lugano, Switzerland. E-mail: amy.murphy@unisi.ch.

Abstract

Distributed content-based publish-subscribe is emerging as a communication paradigm able to meet the demands of highly dynamic distributed applications, e.g., those made popular by mobile computing and peer-to-peer networks. Nevertheless, the available systems implementing this communication model are unable to cope efficiently with dynamic changes to the topology of their distributed dispatching infrastructure, thus effectively hampering applicability in the aforementioned scenarios.

Dealing with topological reconfiguration is a multi-faceted problem. In this paper, we focus on the fundamental issue raised by content-based systems: how to reconcile the information used to route events to subscribers in the face of the topological changes forced by dynamic reconfiguration. Our study begins by analyzing the mechanics of event and subscription propagation in mainstream systems, and their dynamics in the presence of reconfiguration. This analysis enables us to characterize some fundamental phenomena, which are key to defining efficient reconfiguration protocols. The paper defines four such protocols that cover different areas of the tradeoff space defined by applicability and performance. Our solutions are evaluated through simulation showing that indeed they all provide good performance.

I. INTRODUCTION

The publish-subscribe communication model is enjoying increasing popularity, both in research and industry. In this model, application *clients* interact by *publishing* events and by *subscribing* to the classes of events they are interested in. Content-based systems provide a higher level of flexibility by allowing the client to specify these classes using linguistic facilities to match a pattern against event content. A number of content-based publish-subscribe systems are available, differing mainly in the design of the *event dispatcher*, the middleware component responsible for collecting subscriptions and forwarding events to subscribers. In particular, since the first successful centralized implementations, commercial and academic efforts have brought increased scalability by realizing the event dispatcher by means of a distributed architecture, composed of dispatching servers interconnected through an overlay network. Examples of these systems include Siena [8], Jedi [17], Gryphon [4], and many others.

Beyond scalability, the next challenge for distributed publish-subscribe middleware is to tolerate high levels of churn in its distributed dispatching infrastructure. The motivations are numerous. Mobility is increasingly becoming part of mainstream computing. Peer-to-peer applications require very fluid application-level networks for information sharing and dissemination. Companies are frequently undergoing administrative and organizational changes, and so is the

logical and physical network enabling their information systems. 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, the majority of existing *systems* are not up to this task as their static networks of dispatchers are simply unable to operate in dynamic settings.

This problem is particularly important for the large number of content-based systems that seek to exploit the efficiency of tree-based topologies for event dispatching [8]. Even minimal amounts of churn prove, in fact, disastrous for their performance in the absence of adequate mechanisms for *reconfiguring* their operation in response to topology changes. Addressing this issue is therefore the goal of the work we present in this paper. Specifically we consider the routing strategy adopted by the majority of content-based publish-subscribe systems, namely *subscription forwarding*, and propose and evaluate a family of protocols to reconfigure its operation in the presence of churn.

We start by analyzing an existing solution suggested by some authors [8], [51]. This protocol, which we refer to as *"strawman"* due to its inherent simplicity, maintains event routing consistent by properly triggering (un)subscription requests in response to the (dis)appearance of links towards dispatchers. We first demonstrate the inefficiencies of such a naive solution and examine its characteristics. Then, we identify the core ideas around which we structure four new efficient reconfiguration protocols with varying degrees of performance, and applicability. The simplest of our protocols enables overhead reductions up to 50% while retaining the same applicability as the strawman one. More complex protocols, with increasing applicability constraints, instead, yield improvements up to 80%, by exploiting increasing levels of knowledge about how the overlay network changed during reconfiguration.

We carefully evaluate our protocols across a number of dimension by means of extensive simulations that show their scalability with respect to both network size and the frequency reconfigurations. We then integrate this evaluation with a discussion of the tradeoffs between the performance and applicability associated with each protocol. These contributions make our suite of protocols a complete set of solutions that are able to cover the variety of sources of reconfiguration as well as the environments in which content-based publish-subscribe may operate.

The paper is structured as follows. Section II provides the reader with the necessary background

in content-based publish-subscribe systems. Section II-B defines the reconfiguration problem we are tackling in this paper. Section III states some basic assumptions, presents the strawman protocol, highlights its weaknesses, and makes some observations that are at the core of the reconfiguration protocols introduced in this paper and presented in detail in Section IV. Their effectiveness is quantitatively assessed through simulation in Section V, while overall qualitative considerations are drawn in Section VI. Section VII places our work in the context of related efforts. Finally, Section VIII ends the paper with brief concluding remarks.

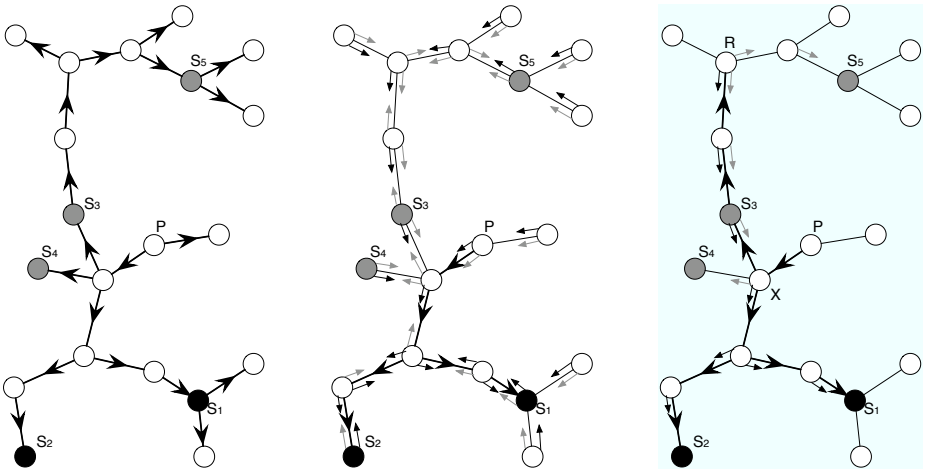
II. Content-Based Publish-Subscribe Systems

The success of the content-based publish-subscribe model results from the expressiveness of its subscription language, which enables clients to select events according to arbitrarily complex *filters* operating on the content of events. This allows it to support a wider range of applications with respect to *subject-based* systems, where subscriptions identify only classes of events belonging to a given channel or subject.

Even if some proposals for alternative approaches have appeared in the literature [9], [44], [6], [41], [32]), a large number of implementations of the content-based paradigm exploit a *subscription forwarding* strategy. This strategy distributes subscription information to establish routes for event dissemination and is therefore particularly suited for scenarios where the frequency of event generation is higher than the frequency at which subscriptions change, which is the most common case in application exploiting the publish-subscribe paradigm.

Despite its success, the subscription forwarding strategy assumes that dispatching servers are laid out on a static tree overlay network, and is therefore particularly vulnerable to scenarios in which the overlay is forced to change by network dynamics. The protocols we present in this paper specifically address this drawback of tree-based publish-subscribe, thereby enabling it to operate in dynamic scenarios ranging from peer-to-peer to mobile ad hoc networks.

In the following, we further specify the focus of our work first by providing a precise description of subscription forwarding and then by abstracting the need to adapt to dynamics in the definition of the corresponding *reconfiguration problem*. It should be noted that while our focus is mainly on content-based systems, both our protocol and the underlying subscription forwarding technique can also be applied to the less challenging case of subject-based publish-subscribe.



(a) Event forwarding (b) Subscription forwarding (c) Hierarchical forwarding

Fig. 1. Publish-subscribe routing schemes. Circles denote dispatchers. Filled circles represent dispatchers subscribed to a given pattern. Arrows outgoing from a dispatcher denote the content of its subscription table. Thick lines and arrows show the path followed by the "black" event published by P. Clients are not shown to avoid cluttering the figure.

A. Subscription Forwarding

In *Subscription forwarding*, a set of *dispatching servers*[1] (see Figure 1) are interconnected as an overlay tree and cooperatively establish routing paths for events by spreading knowledge about the subscriptions they receive. When a dispatchers receives a subscription from a client, it first inserts it into its own subscription table together with the identifier of the subscriber. Then, it also forwards it to all the neighboring dispatchers. During this propagation, each dispatcher behaves as a subscriber with respect to its neighbors, each of which records the subscription in its table and re-forwards it to its neighbors, except to the one that sent it. This scheme is usually optimized by avoiding subscription forwarding of the same pattern in the same direction[2]. This process effectively sets up the routes that a published event follows in its journey from a publisher to a subscriber. Requests to unsubscribe are handled and propagated analogously to subscriptions, although at each hop an entry in the subscription table is removed rather than inserted. Figure 1(b) illustrates the concept graphically.

[1]Hereafter we refer to *dispatching servers* simply as *dispatchers*.

[2]Other optimizations are possible, e.g., by defining a notion of "coverage" among subscriptions, or by aggregating them, as in [8], [24], [47], [10].

B. The Reconfiguration Problem

The reconfiguration problem we address can be defined informally as: *the need to adapt the dispatching infrastructure of a distributed publish-subscribe system to changes in its topology, without interrupting the normal system operation*. In the case of subscription forwarding, this can be broken down into three sub-problems, namely:

1) repairing the overlay network supporting the dispatching infrastructure, to retain connectivity among dispatchers without creating loops;
2) reconciling the subscription information held by each dispatcher and used for routing messages, to keep it consistent with the topological changes above without interfering with the normal processing of subscriptions and unsubscriptions;
3) recovering messages lost during reconfiguration.

In this paper, we propose several solutions for correctly reconfiguring the subscription information, i.e., for the second of the aforementioned problems, and compare them with respect to both their performance and their applicability. This is motivated by the fact that *maintaining the consistency of subscription information is the defining problem of content-based routing for publish-subscribe systems*. If the information necessary for event dispatching is misconfigured, or propagated inefficiently, the whole purpose of a content-based system is undermined.

Clearly, the two remaining problems are also relevant, but their solutions heavily depend on the specific application scenarios being considered. Moreover, we have addressed both of them in previous work [14], [15], [31], [25] with protocols that can be readily combined with those we present in this paper. This allows us to treat the three problems as orthogonal, by tackling each one in isolation and minimizing their interactions. Specifically, in the following we only assume that the dispatching tree is kept connected and cycle-free by another module as detailed in the next section.

III. Assumptions, Baseline, and Fundamental Concepts

With this understanding of the reconfiguration problem, we now illustrate the foundations our protocols rely upon. We begin by stating in Section III-A the assumptions we make about the configuration of the system and the architecture of the publish-subscribe middleware. Then, in Section III-B we present the *strawman protocol*, a basic solution [8], [51] which serves as a baseline for the results presented in this paper. Finally, in Section III-C we analyze the behavior

of the strawman protocol, highlight its drawbacks, and derive the fundamental observations that inspired the design of the reconfiguration protocols we present in Section IV.

A. Basic Assumptions

In addition to considering a content-based publish-subscribe system using subscription forwarding on an unrooted tree overlay network, hereafter we make the following assumptions.

Communication. 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, e.g., by using TCP for communication between dispatchers.

Reconfigurations. We focus on reconfigurations where a single link is added or removed, as this is the most general modification of the network topology. Indeed, a dispatcher failure or appearance can be modeled as several links simultaneously vanishing or appearing. Although this assumption may preclude some optimizations, by considering the most general case we focus on the essence of the reconfiguration problem and unveil its fundamental characteristics and tradeoffs.

Abstract middleware architecture. Based on the discussion at the end of Section II-B, we assume the presence of two distinct modules: the *routing module* and the *tree maintenance module*. The routing module manages the reconciliation of routing information upon topological changes, and any of the protocols described in this paper can provide an implementation for this module. Instead, the tree maintenance module takes care of updating the tree overlay network, and notifies the routing module of any change, as discussed next.

Interface between tree maintenance and routing. At a minimum, the tree maintenance module must notify the routing module at each end-point of a broken or new link, so that it can take appropriate actions. These notifications are independent (i.e., dispatchers on the old link do not know the identities of the dispatchers on the new link and vice versa) and can be easily implemented locally. Some of our protocols make more stringent assumptions on the underlying tree maintenance module. In these cases, we state these assumptions as part of the description of the different protocols in Section IV, and analyze their impact in Section VI.

B. Baseline: The Strawman Protocol

In this section we describe the basic solution found in the literature [8], [51], which we refer to as the STRAWMAN protocol. This protocol provides an ideal starting point for our investigation for three main reasons. First, it serves as a base for understanding reconfiguration and therefore how the process can be improved. Second, it serves as a point of comparison for demonstrating the improvements achieved by our protocols. Third, it uses only the normal publish-subscribe operations, allowing us to introduce, in a simple context, the notation that we use later for describing more complex protocols.

The operation of the STRAWMAN protocol can be summarized as follows:

- When a link breaks, messages can no longer be sent across it. All subscriptions received previously along that link are useless and should be removed. Therefore, when a dispatcher is notified that a link towards a neighbor n is broken, it behaves as if it received an unsubscription message for all the subscriptions previously sent by n. This removes the routes forwarding events across the broken link.
- When a new link appears, message forwarding across it must be properly established, by propagating the appropriate subscriptions. Therefore, when a dispatcher is notified of a new neighbor n, it sends to n subscription messages for all of the patterns in its subscription table, thus enabling the forwarding of matching events across the new link.

The key to the STRAWMAN protocol is that the operations above are accomplished entirely with normal subscription and unsubscription messages. These propagate normally along the tree, updating routing information along the way.

Figure 2 shows the pseudo-code executed on a dispatcher for these reconfiguration operations, along with the normal subscription, unsubscription, and event processing. We assume that each dispatcher knows its neighbors, maintained in the $neighbors$ variable. Also, we assume that each dispatcher holds a subscription table $subTab$ containing subscriptions in the form $\langle n, p \rangle$, to record that the neighboring dispatcher n is subscribed to pattern p. The behavior of clients is not modeled explicitly as it does not directly affect our reconfiguration protocols, which instead focus on inter-dispatcher routing. Moreover, in the scenarios we target (e.g., mobile ad hoc and peer-to-peer networks) the publish-subscribe system is likely to be deployed so that clients and dispatchers are co-located [27].

```
//Process event from neighbor n
eventReceived(n, EVENT)
  send EVENT to matching subscribers except n

//Unsubscription received from neighbor n
unsubscriptionReceived(n, UNSUB(p))
  if ⟨n, p⟩ ∈ subTab then
    processUnsubFrom(n, p)

//Subscription received from neighbor n
subscriptionReceived(n, SUB(p))
  if ⟨n, p⟩ ∉ subTab then
    subTab ← subTab ∪ {⟨n, p⟩}
    if this is first subscription
          received for p then
      send SUB(p) to all neighbors except n
    else if this is second subscription
          received for p then
      send SUB(p) to first subscriber

//Remove link to neighbor n
removeLinkTo(n)
  for all ⟨n, p⟩ ∈ subTab do
    processUnsubFrom(n, p)                    [1]
  neighbors ← neighbors − {n}

//Add link to new neighbor n
addLinkTo(n)
  neighbors ← neighbors ∪ {n}
  for all ⟨n, p⟩ ∈ subTab do
    send SUB(p) to n                          [2]

//Process unsub. for pattern p from neighbor n
processUnsubFrom(n, p)
  subTab ← subTab − {⟨n, p⟩}
  if no more subscribers for p in subTab then
    send UNSUB(p) to all neighbors except n
  else if only one subscription
          ⟨n′, p⟩ in subTab then
    send UNSUB(p) to n′
```

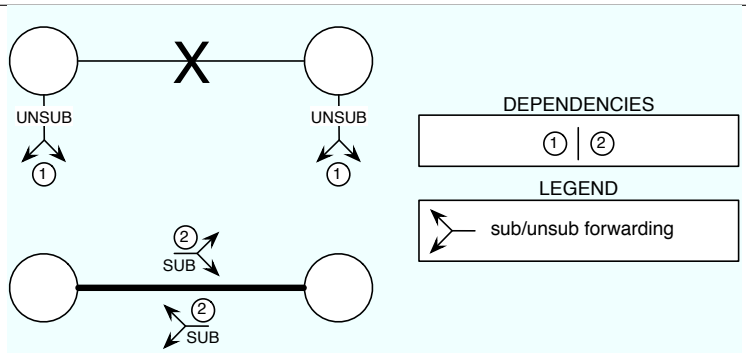
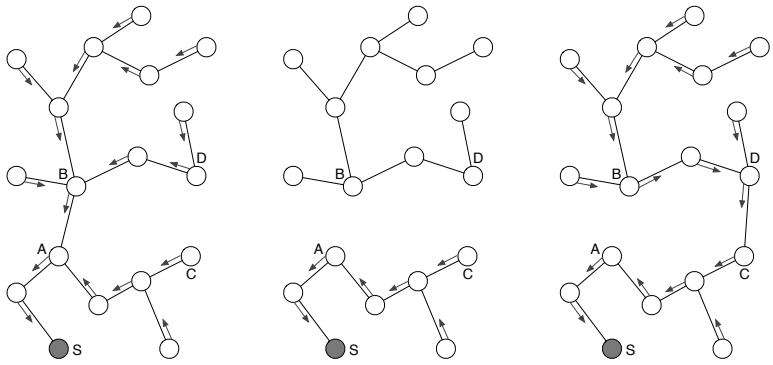


Fig. 2. Pseudo-code and schematic of the STRAWMAN protocol. The pseudo-code also outlines normal event, subscription and unsubscription processing. Numbers in square brackets indicate the corresponding messages in the schematic below. The schematic assumes the top link is removed and the bottom link is added. To the right of the schematic is a dependence diagram showing implied (but not strict) dependence among steps with numbering. Also, the legend shows that only normal subscription and unsubscription messages are sent during the execution of the protocol, with the split arrow indicating propagation of the message along the tree.

Each operation in the pseudo-code executes local to a dispatcher. Moreover, only one operation at a time can be executed. The operations eventReceived, subscriptionReceived, and unsubscriptionReceived are triggered by the arrival of the corresponding messages at the dispatcher, while removeLinkTo and addLinkTo are called by the tree maintenance module to notify a dispatcher of the removal or appearance of a link towards one of its neighbors. The identifier of the dispatcher where an operation is executing is obtained from the variable $self$.

The figure also introduces a simple graphical notation to represent the protocol behavior,



(a) before (b) during (c) after

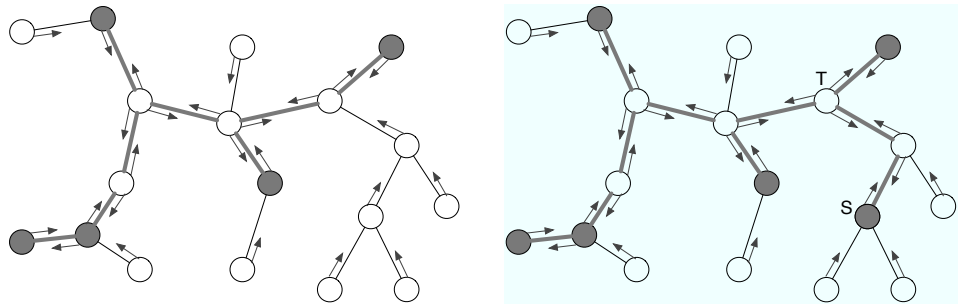
Fig. 3. A dispatching tree of before, during and after a reconfiguration performed using the STRAWMAN approach.

whose usefulness will be appreciated when we discuss more complex protocols later on. The picture on the left represents the two end-points of the broken link (top) and those of the new link (bottom), and shows pictorially which messages are being sent, where, and how—in this case, unsubscription messages sent by the end-points of the old link and subscription messages sent by the end-points of the new link, both propagating on the tree as usual. The schematic also shows the dependencies between these messages. In this case, the dependence diagram in the middle shows that the sending of unsubscriptions and subscriptions, respectively numbered as 1 and 2, can happen concurrently. In the protocols introduced later, sequential dependencies, depicted by arrows, are also introduced.

C. Understanding Propagation and Reconfiguration

In this section we highlight a fundamental problem of the STRAWMAN protocol, and state several observations that are at the core of the reconfiguration protocols we introduce in Section IV.

The Fundamental Problem: *Subscriptions are removed and immediately re-inserted.* The STRAWMAN protocol is the most natural protocol when reconfiguration involves only an isolated link insertion or removal. However, the most frequent case in a dynamic network is one where a broken link is quickly replaced by a new one. In this case, the STRAWMAN protocol is highly inefficient, as illustrated in Figure 3. If the unsubscriptions propagate throughout one of the sub-trees before the subscriptions start (which is the most likely case), the effect is that many



(a) A pattern tree (b) S issues a subscription

Fig. 4. Understanding the propagation of subscriptions. In the left figure, the pattern tree for a gray pattern is shown. In the right figure, when a dispatcher S issues a subscription for the gray pattern, its subscription is propagated only up to the closest dispatcher (T) that is part of the pattern tree.

of the subscriptions in this sub-tree are removed only to be re-added after a short time. This phenomenon generates unnecessary overhead, whose negative impact is proportional to the size of the system and the degree of reconfiguration. Providing alternative protocols that are not affected by the same problem, and therefore achieve a considerable overhead reduction over STRAWMAN, is the purpose of the work described in this paper. Before delving into the details of the protocols, illustrated in the next section, we make some observations that provide the foundations of their design.

Observation 1*: The higher the density of subscribers, the shorter the propagation of subscriptions.* To understand this observation, it is useful to analyze how subscriptions propagate on the dispatching tree. Let us define the *pattern tree* for a given pattern p as the (minimal) sub-tree of the dispatching tree connecting all the dispatchers subscribed to p. Figure 4(a) visualizes the concept by showing the pattern tree for a "gray" pattern.

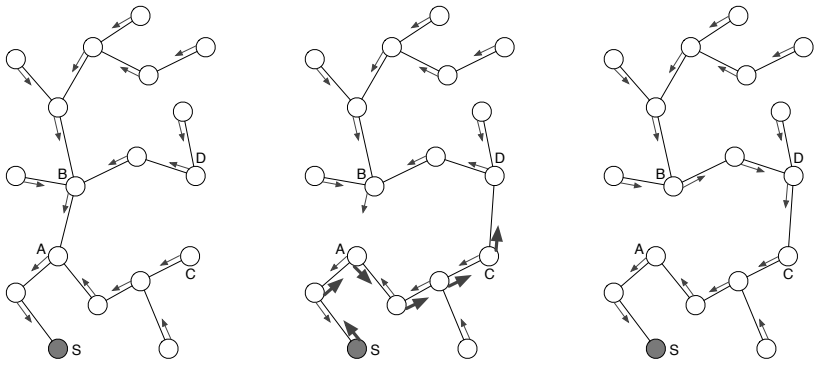
Based on this definition, the following rule holds for systems based on the subscription forwarding strategy outlined in Section II: *a subscription for a pattern p is propagated along the unique route up to the pattern tree for p, if it exists; to the whole tree, otherwise.* Clearly, if the new subscriber for p already lies on the pattern tree for p no subscription needs to be propagated. A similar rule holds for unsubscriptions: *an unsubscription for a pattern p propagates up to the closest dispatcher that, after having rearranged its subscription table by processing the unsubscription message, remains part of the pattern tree.*

To understand these rules we observe that the routing tables of the dispatchers belonging to the pattern tree for p are organized in such a way that any event matching p that reaches one of these dispatchers is forwarded to all the others—i.e., it is broadcast along the pattern tree. This is evident in Figure 4(a), where each link of the pattern tree has event routes (represented by arrows) in both directions. Instead, the routing tables of the dispatchers outside the pattern tree are set so that they route the events matching p towards the pattern tree but not vice versa, i.e., once events reach the pattern tree they are never forwarded outside of it. Again, this is visualized in Figure 4(a). The mechanics of propagation are easily understood by focusing on what happens when a new subscriber S appears. Clearly, if no other subscriber exists, the subscription is simply broadcast to the rest of the tree, as discussed in Section II. Instead, if a pattern tree has already been established (even with a single subscriber), as in Figure 4(b), the subscription is propagated only up to the closest dispatcher belonging to it, e.g., T in the figure. Effectively, the propagation of this new subscription establishes the bidirectional routes that extend the pattern tree and enable the broadcast of matching events towards the new subscriber. Similar considerations hold for unsubscriptions.

These rules prompt two considerations. 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 dispatcher in the pattern tree, unless there are no subscribers. Second, as more subscriptions are added, the size of the pattern tree increases, thus shortening the path traveled by subsequent subscriptions. Unsubscriptions, on the other hand, decrease the number of dispatchers in the pattern tree so that subsequent subscriptions and unsubscriptions must propagate to larger sections of the dispatching tree.

These considerations lead to a criterion for designing reconfiguration protocols: keep the tree "dense" of subscriptions, and thus reduce the overhead caused by the propagation of subscriptions. This is naturally accomplished by performing subscriptions before unsubscriptions, essentially reversing the normal sequence of operations of the STRAWMAN protocol.

Observation 2*: Subscriptions across the old link may not require propagation.* Our second observation comes from analyzing the reconfiguration process that involves the new link. In the STRAWMAN protocol, the end-points of the new link simply send subscriptions for all the entries in their subscription table, to ensure that all the events of interest generated in the other sub-tree



(a) before (b) during (c) after

Fig. 5. Extraneous subscriptions created during reconfiguration when subscriptions precede unsubscriptions.

are properly forwarded. While this is sufficient, however, it is not entirely necessary.

Consider, for example, the situation shown in Figure 5 in which only one dispatcher is subscribed to a particular pattern. It may happen that the unsubscription issued by dispatcher B, that will eventually remove the arrows between D and B, propagates slowly and reaches D only after the new link has opened and D has exchanged its subscriptions with C. This causes the insertion of extraneous subscriptions (the thick arrows in Figure 5(b)), which will be eventually removed by the slowly propagating unsubscriptions issued by B (Figure 5(c)). Therefore, the protocol still correctly restores the routing information, but it does so in an inefficient way. This phenomenon can occur in the STRAWMAN protocol, but it is even more likely to occur if the subscription and unsubscription operations are reversed, as suggested above.

The key observation is that the link between C and D is not being added in isolation, but rather in response to the removal of the link between A and B. By scrutinizing the subscriptions on these latter dispatchers, we can decide which subscriptions should be exchanged between C and D and, equally important, which should not. Specifically, any subscription that is present on the old link and serves *only* to route events to the other sub-tree (e.g., the one at B towards A) should *not* be propagated across the new link. This is sufficient to prevent the extraneous subscriptions shown in Figure 5(b).

Observation 3*: The impact of reconfiguration is limited to a well-defined path.* While the previous observations may help in reducing unnecessary subscriptions, our next observation focuses on

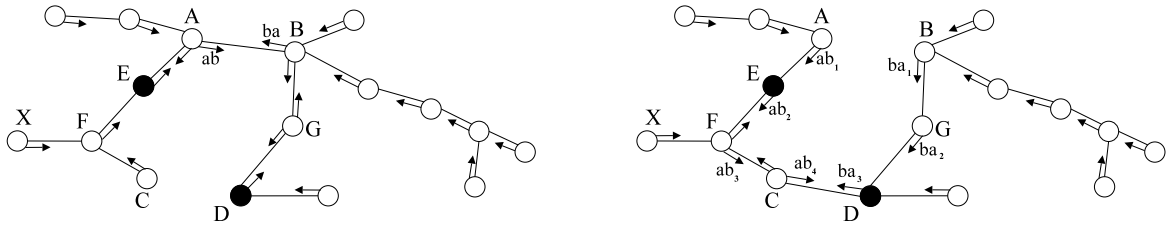


Fig. 6. A dispatching tree before and after a reconfiguration, showing explicitly the subscriptions that replace the broken link.

narrowing the scope of reconfiguration in terms of dispatchers involved, and therefore helps in designing protocols that limit its impact. To find which dispatchers are involved we note that, from the perspective of event routing, the events that were intended to traverse the vanished link must be re-routed across the new link to reach the other part of the tree. We therefore observe that only the subscription tables of the dispatchers on the path between the old and new link need to be changed. All the other dispatchers simply forward events to this path, and remain unchanged during reconfiguration. We refer to this path as the *reconfiguration path* and define it as the concatenation of two sequences of dispatchers:

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

Figure 6 shows a reconfiguration example where the link (A, B) is being substituted with the link (C, D). In this case, (A, E, F, C) is the head path and (D, G, B) is the tail path, yielding the reconfiguration path (A, E, F, C, D, G, B). As a result of the reconfiguration, the subscription ab, which was exploiting the vanished link (A, B) to route events towards B's sub-tree, is removed by the reconfiguration and it is replaced by subscriptions ab_1, ab_2, ab_3, and ab_4. Similarly, the effect formerly achieved by ba is obtained by ba_1, ba_2, and ba_3.

From the example, we can derive two considerations that help in understanding the mechanics of reconfiguration. First, a subscriber's sub-tree always contains complete routing information to

allow events to reach the subscriber from any of its dispatchers. Second, some of the subscriptions necessary to allow events to reach the other sub-tree may already be present due to other subscribers. In this example, in fact, only ab_2, ab_3, and ab_4 need to be added in A's sub-tree: ab_1 was already present to route events from A towards the subscriber E. Similarly, in the other sub-tree, only ba_3 needs to be added towards A, since ba_1 and ba_2 were already present because of D.

These considerations allow us to derive a general principle: the subscriptions replacing those on the first end-point of the old link (e.g., from A to B in Figure 6) are needed only on the head path. Similarly, the subscriptions replacing those on the other end-point of the old link (e.g., from B to A) are needed only on the tail path. No other dispatcher is affected by the reconfiguration.

IV. Four New Reconfiguration Protocols

Based on the previous observations, we have designed four reconfiguration protocols called: Deferred Unsubscription (in two variants Timed Deferred Unsubscription and Notified Deferred Unsubscription), Informed Link Activation, and Reconfiguration Path. Each protocol makes different assumptions about the underlying tree maintenance module. For example, the Timed Deferred Unsubscription protocol retains the assumptions of Strawman, while the Reconfiguration Path protocol assumes that the notification sent by the tree maintenance module to the routing module (see Section III-A) contains the list of dispatchers on the reconfiguration path. The protocols also differ in terms of complexity and of the performance improvement they achieve with respect to the Strawman protocol. The combination of overhead reduction capability, protocol complexity, and assumptions about the tree maintenance module provide the evaluation criteria to decide which protocol to use in a particular environment.

In the following, we provide a detailed description of the protocols. Three of them have been introduced in our previous work [20], [18], [37]. Here we introduce the new Informed Link Activation protocol, and also extend our previous work by *i)* presenting all the protocols in an integrated and detailed way, with a uniform description in terms of informal pseudo-code, *ii)* exhaustively comparing the protocols against one another through simulation in Section V, and *iii)* comparing them qualitatively in Section VI.

A. Deferred Unsubscription

As discussed in Section III, the main drawbacks of the STRAWMAN protocol result from the fact that the unsubscription process initiated by a link removal and the subscription process handling link insertion proceed completely in parallel. Our DEFERRED UNSUBSCRIPTION protocol is based on Observation 1 from Section III-C: keeping the tree dense of subscribers can reduce the overhead of subscription propagation. The protocol leverages off the conventional subscription and unsubscription operations as in the STRAWMAN protocol, but performs 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 deferred. This strategy does not introduce any special mechanism to limit its scope to the reconfiguration path and therefore it may add subscriptions that must be removed immediately after. However, these subscriptions propagate only up to the corresponding pattern tree—a distance likely to be short when the tree is dense of subscriptions. It is worth noting that the reconfiguration described by this protocol 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.

In the following we describe two variants of this protocol, which differ in the mechanism used to defer unsubscriptions.

1) Timed Deferred Unsubscription: In our first and simplest variant of the DEFERRED UNSUBSCRIPTION protocol, shown in Figure 7, the delay is provided by an unsubscription timer T_u that is initialized by each end-point dispatcher when a link breaks. The expiration of this timer triggers the propagation of unsubscriptions, therefore we refer to our protocol as TIMED DEFERRED UNSUBSCRIPTION. Ideally, the delay induced by the timer should coincide with the time needed by the underlying tree maintenance module to restore the connectivity of the tree, plus the time required to propagate subscriptions. As in the STRAWMAN protocol, TIMED DEFERRED UNSUBSCRIPTION only requires the underlying tree maintenance module to notify the end-points of the old and new links.

Links Sharing a Dispatcher. This protocol provides significant advantages over STRAWMAN. However, from Observation 2 in Section III-C we know that delaying unsubscriptions may lead to an unnecessary propagation of subscriptions across the new link. Unfortunately, in this TIMED

```
//Remove link to neighbor n
removeLinkTo(n)
  rId ← newRId()
  manageBrokenLink(n, rId)

//Add link to new neighbor n
addLinkTo(n)
  neighbors ← neighbors ∪ {n}
  for all ⟨n', p⟩ ∈ subTab, n' ≠ n do
    send SUB(p) to n                                   [2]

//Unsubscription timer expires, n is old neighbor,
//rId is the reconfiguration identifier
unsubTimerExpires(n, rId)
  for all ⟨n, p⟩ ∈ pending[rId] do
    processUnsubFrom(n, p)                             [3]
  pending ← pending − pending[rId]

//Manage broken link to n for reconfiguration rId
manageBrokenLink(n, rId)
  if |neighbors| ≥ 1 then
    // self is a non-leaf dispatcher
    pending[rId] ← subTab[n]
    subTab ← subTab − subTab[n]
    neighbors ← neighbors − {n}
    start unsubTimer(n, rId)                           [1]
  else
    // self is a leaf dispatcher
    subTab ← subTab − subTab[n]
    neighbors ← neighbors − {n}
```

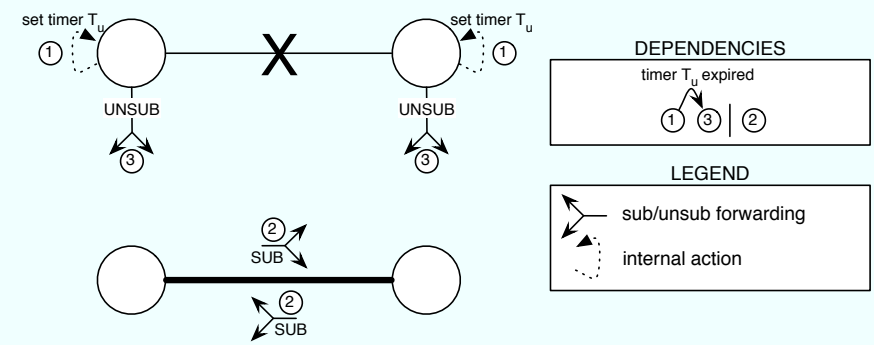


Fig. 7. TIMED DEFERRED UNSUBSCRIPTION pseudo-code and schematic. Arrows in the dependence diagram show strict dependence.

DEFERRED UNSUBSCRIPTION protocol, we assume that the underlying tree maintenance module does not provide any association between the old and new links, making the optimization outlined in Section III-C impossible.

However, in the case where one end-point of the new link coincides with an end-point of the old link we *do* have sufficient information to prevent unnecessary subscription forwarding. While this may at first seem to be a special case, it is actually quite common for overlay network protocols to make this choice. One of the end-points of a removed link is usually responsible for actively repairing the tree and very often also becomes an end-point of the link added during this process.

To handle this case effectively, we simply prevent the reconfiguration from forwarding subscriptions directed *only* towards dispatchers connected through links that are now broken. All

other subscriptions, i.e., those coming from clients attached to the shared dispatcher and those associated to intact links, are propagated as usual. This behavior is evident by looking at the action manageBrokenLink(n, rId) in Figure 7, which is invoked when the link between the current dispatcher and n breaks. When this happens, the set of patterns associated to the neighbor n (denoted by $subTab[n]$) are immediately removed from the local subscription table and stored in a separate $pending$ table. This way they are ignored both when processing other concurrent (un)subscriptions and when propagating subscriptions to newly added links. On the other hand, when the timer expires the unsubscriptions for the patterns in the $pending$ table are propagated, as required by the protocol.

Leaf Dispatchers. A special case in which the new and old links share an end-point is when a leaf dispatcher is detached and re-attached to a different dispatcher. This case is fairly frequent because leaf dispatchers are usually a large fraction of the total number of dispatchers and, since they are at the fringe of the system, they are more subject to reconfiguration.

The peculiarity of this case is that deferring unsubscriptions becomes superfluous in the case of a detached leaf dispatcher. A detached leaf dispatcher has, by definition, no neighbors to send messages to. As a result it can unsubscribe locally without updating its $pending$ table and without setting timers. This optimization not only simplifies processing when a leaf dispatcher is involved in a reconfiguration, but also allows the protocol to avoid the propagation of unnecessary unsubscriptions across the new link when the timer expires.

2) Reducing the Dependence on Timers: Notified Deferred Unsubscription: In the TIMED DEFERRED UNSUBSCRIPTION protocol the timer plays a crucial role. If its value is too small, the overhead of the protocol approaches that of STRAWMAN because unsubscriptions are triggered too early, before subscriptions have been restored. If it is too large, obsolete routes remain in place and steer events where there are no subscribers, thus increasing overhead.

Although it is possible to determine experimentally an appropriate timer value for a specific system (as we did in the experiments presented in Section V), we target a dynamic environment where reliance on a statically set timer is inherently approximate. Rather than trying to dynamically adjust the timer, our next protocol complements the timer with a deterministic notification process while still maintaining the benefits of deferring unsubscriptions. For this protocol, NOTIFIED DEFERRED UNSUBSCRIPTION, we assume that the tree maintenance module

```
//Remove link to neighbor n,
//rId is the reconfiguration identifier
removeLinkTo(n, rId)
  TimedDefUnsub.manageBrokenLink(n, rId)                    [1]

//Add link to new neighbor n
addLinkTo(n, rId)
  TimedDefUnsub.addLinkTo(n)                                [2]
  send FLUSH(rId) to n                                      [3]

//Unsubscription timer expires, n is old neighbor,
//rId is the reconfiguration identifier
unsubTimerExpires(n, rId)
  TimedDefUnsub.unsubTimerExpires(n, rId)                   [4]

//FLUSH message received from neighbor n
flushReceived(n, FLUSH(rId))
  if self has unsubTimer(n', rId) then
    //n' is other end-point of old link
    cancel unsubTimer(n', rId)
    TimedDefUnsub.unsubTimerExpires(n', rId)                [4]
  else
    //self is not end-point of old link
    send FLUSH(rId) to all neighbors except n               [3]
```

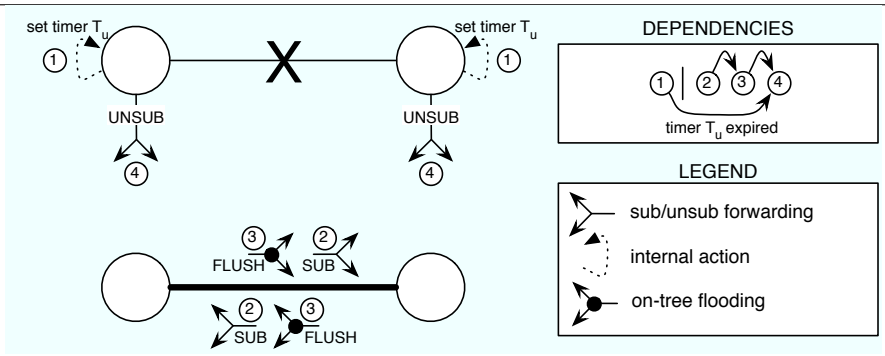


Fig. 8. NOTIFIED DEFERRED UNSUBSCRIPTION pseudo-code and schematic. In the dependence diagram, a step with more than one incoming arrow indicates it can be triggered by either of the previous steps.

is able to associate the removed link with the inserted one by assigning a unique identifier to each reconfiguration. We also assume that the calls to removeLinkTo are made *before* the calls to addLinkTo, thus ensuring the proper tagging of links as broken before subscriptions are forwarded[3]. These assumptions make it possible to implement the aforementioned notification process by means of a FLUSH message, sent by the end-points of the new link towards the corresponding end-points of the old link after the former finish propagating subscriptions. Since we assume FIFO links, when the end-points of the old link receive the FLUSH message they can correctly deduce that the subscriptions from the new link have propagated all the way to the old link, and therefore the unsubscription process can start. This behavior is outlined in Figure 8.

The remainder of the processing is identical to the TIMED DEFERRED UNSUBSCRIPTION protocol—including the presence of the unsubscription timer. In fact, although the FLUSH mes-

[3]Both assumptions are satisfied by the overlay maintenance protocols defined in our research, i.e., [31] and [25].

sage serves the same purpose of the timer (i.e., to start the propagation of the unsubscriptions) it is possible that, due to concurrent reconfigurations, the FLUSH message will not reach the end-points of the old link. In this case, the unsubscriptions begin propagating when the timer expires. To improve readability we use the notation protocol.operation (e.g., TimedDefUnsub.addLinkTo in Figure 8) to reuse operations defined in a protocol previously presented.

It is worth noting that, for simplicity, the FLUSH message is broadcast along the entire tree, although it only needs to propagate along the reconfiguration path to reach the end-points of the old link. There are two ways to optimize this propagation. One is to exploit information about the reconfiguration path if it is provided by the tree maintenance module. The other is to exploit the propagation of the subscriptions sent by end-points of the new link to guide the propagation of the FLUSH. More specifically, the FLUSH can be piggybacked on these subscriptions as they propagate towards the removed link; this way, the FLUSH only needs to be broadcast from the point where the last subscription stops propagating. The benefits arising from these optimizations should, nevertheless, be weighed against the complexity they introduce and against the fact that broadcasting the FLUSH message along the whole tree is likely to be more resilient to concurrent reconfigurations.

B. Informed Link Activation

In Section III-C, we pointed out that unnecessary reconfiguration overhead comes primarily from two sources. First, according to Observation 1, unsubscriptions from the old link may propagate unnecessarily and temporarily remove routes that are needed after the reconfiguration. Second, according to Observation 2, subscriptions from the new link may propagate stale routing information that was only needed before the reconfiguration occurred. Both DEFERRED UNSUBSCRIPTION protocols address the first problem by postponing the propagation of unsubscriptions until the subscriptions from the new link have finished propagating. However, they only address the second problem when the new and old links share an end-point. The shared end-point identifies the subscriptions that are directed only along the vanished link and avoids propagating them.

Our new INFORMED LINK ACTIVATION protocol extends the behavior of the DEFERRED UNSUBSCRIPTION protocols by addressing Observation 2 even when the end-points of the new and old links are not shared. To accomplish this it propagates information about unnecessary

```
//Substitute link (self, o) with (n1, n2)
//rId is the reconfiguration identifier
substituteLinkTo(o, n1, n2, rId)
  //n1 is in same sub-tree as self
  P ← {p | only o is subscribed to p}
  if |neighbors| ≥ 1 then
    // self is a non-leaf dispatcher
    start unsubTimer(o, rId)                          [1]
  else
    // self is a leaf dispatcher
    subTab ← subTab − subTab[o]
  neighbors ← neighbors − {o}
  send ACTIVATE(rId, P, n2) to n1                     [2]

//Unsubscription timer expires
//n is old neighbor,
//rId is the reconfiguration identifier
unsubTimerExpires(n, rId)
  for all ⟨n, p⟩ ∈ subTab do
    processUnsubFrom(n, p)                            [6]

//Subscription timer expires
subTimerExpires(n, rId)
  for all p ∈ taggedSubs[rId] do
    if ⟨p, n'⟩ ∈ subTab and n' ≠ n then
      send SUB(p) to n                                [7]
  cancel subTimer(n, rId)
  taggedSubs[rId] = ∅

//FLUSH message received
flushReceived(n, FLUSH(rId))
  if self has unsubTimer(n', rId) then
    //n' is other end-point of old link
    cancel unsubTimer(n', rId)
    unsubTimerExpires(n', rId)                        [6]
  else
    //self is not end-point of old link
    send FLUSH(rId) to all neighbors except n         [5]

//ACTIVATE message received
activateReceived(ACTIVATE(rId, P, n))
  neighbors ← neighbors ∪ {n}
  start subTimer(n, rId)                              [3]
  for all p ∈ subTab do
    if p ∉ P or ⟨p, self⟩ ∈ subTab then
      send SUB(p) to n                                [4]
    else
      taggedSubs[rId] ← taggedSubs[rId] ∪ {p}
  send FLUSH(rId) to n                                [5]
```

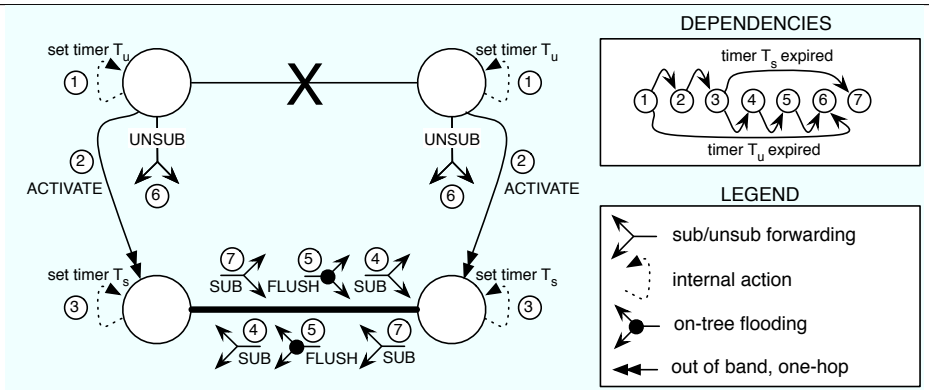
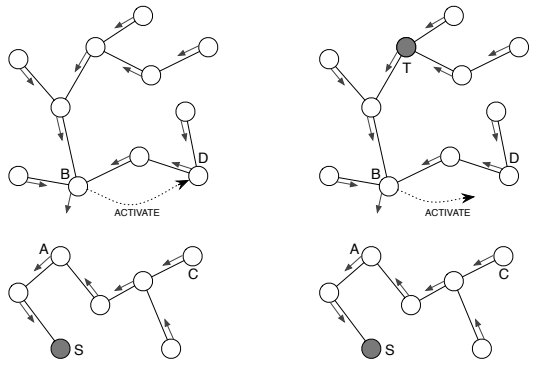


Fig. 9. INFORMED LINK ACTIVATION pseudo-code and schematic.

subscriptions from the old link to the new one, allowing the end-points of the new link to recognize these subscriptions and avoid their propagation. The protocol is based on the same assumption as NOTIFIED DEFERRED UNSUBSCRIPTION, namely, that the tree maintenance module is able to associate the new link with the old one. Moreover, the communication between the end-points of the old and new link can occur either by sending a direct, out-of-band ACTIVATE message or by piggybacking this information on messages sent by the tree maintenance module.



(a) Activating a link. (b) Concurrent subscriptions.

Fig. 10. Two scenarios in the INFORMED LINK ACTIVATION protocol.

Figure 9 shows the pseudo-code and schematics for the protocol. We assume that the tree maintenance module invokes the operation substituteLinkTo on each of the old link end-points when a reconfiguration occurs, informing them of the identity of the end-points of the new link. In this operation, each end-point of the old link determines the set of patterns P that are used to route events only toward the other sub-tree and propagates P to the corresponding end-point (i.e., in the same sub-tree) of the new link by means of an ACTIVATE message (step 2 in the pseudo-code). Upon receiving the ACTIVATE message, the end-points of the new link propagate their subscriptions across the new link (step 4). The processing is similar to the addLinkTo operation found in the previous protocols, except here a subscription is propagated across the new link only if it is local or it does not belong to the set of patterns P contained in the ACTIVATE message. For instance, Figure 10(a) shows a situation similar to the one depicted in Figure 5 where, upon breakage of the link (A, B), B sends to D an ACTIVATE message where P contains the pattern for B's subscriptions towards A. Upon receiving this message, D does not propagate the subscriptions in P across the new link, therefore avoiding the generation of the extraneous subscriptions shown in Figure 5.

Unsubscriptions can be dealt with as in the DEFERRED UNSUBSCRIPTION protocols. In Figure 9 (and later in the simulations of Section V) we use the technique described in NOTIFIED DEFERRED UNSUBSCRIPTION. Unsubscriptions are triggered at the broken link by the receipt of a FLUSH message sent by the end-points of the new link (step 5) or, if this does not propagate

fast enough, by the expiration of a timer (set in step 1). Moreover, as in the DEFERRED UNSUBSCRIPTION protocols, leaf dispatchers that lose the link to their only neighbor do not wait for the timeout and process the corresponding unsubscriptions as soon as the link is removed.

The processing just described is sufficient when the only subscriptions and unsubscriptions being propagated are those determined by a reconfiguration. In reality, however, subscriptions can change concurrently to a reconfiguration, and multiple reconfigurations can occur in parallel. In these cases, some of the subscriptions that have been deemed unnecessary may still need to be propagated. For instance, consider the situation depicted in Figure 10(b). Dispatcher T may decide to subscribe to the same "gray" pattern of interest to S, concurrently to the replacement of link (A, B) with (C, D). In this case, contrary to what we stated earlier, D should forward the subscription even if it is contained in P. Interestingly, D has no way to know whether a subscription is used only by B or also by some other dispatcher in its sub-tree. This becomes evident only after the unsubscriptions eventually issued by B have propagated to D, and have purged unnecessary entries from its subscription table.

To address the issue, the INFORMED LINK ACTIVATION protocol uses an additional *subscription timer* T_s, started upon receipt of the ACTIVATE message (step 3). The expiration of this timer (step 7) causes the propagation of those subscriptions in P that were not propagated in step 4 but that are still in the subscription table. These are contained in the variable $taggedSubs$ in Figure 10(b). In the example of Figure 10, this technique leads to the correct propagation of the subscription issued by T, as it is not removed by the unsubscriptions propagated by B. Both the timer and the subscriptions in $taggedSubs$ are associated with a reconfiguration identifier rId, to distinguish among multiple reconfigurations.

Clearly, the value of the subscription timer must be large enough to allow the unsubscriptions generated at the old link to be propagated to the new link before it expires. If t_{rep} is the sum of the time required by the tree maintenance module to locate a new route plus the time required for the propagation of the ACTIVATE messages, and t_{prop} is the time required for the propagation of unsubscriptions from the old to the new link, then to maximize performance the values of the unsubscription and subscription timers, T_u and T_s, should satisfy the following constraint:

$$T_u + t_{prop} < T_s + t_{rep} \tag{1}$$

It is worth observing that the use of the subscription timer may increase the delay experienced

by clients before receiving events after a new subscription. However, this is a small price to pay with respect to the great reduction in overhead achieved by the protocol.

C. Reconfiguration Path

Although the previous protocols keep the tree dense of subscriptions to limit the scope of the reconfiguration, it is possible that subscriptions (and possibly unsubscriptions) propagate beyond the reconfiguration path, introducing unnecessary overhead. Our next protocol focuses explicitly on the dispatchers on the reconfiguration path. We assume the latter is computed by the tree maintenance module and communicated to the content-based routing module. The protocol operates in a strictly sequential way by propagating a special reconfiguration message along the reconfiguration path, from one end-point of the broken link to the other. Upon receiving this reconfiguration message, dispatchers rearrange their subscription table to take into account the changes in the overlay topology. This sequential way of operating reduces the overhead to a minimum, but is also the source of the main weakness of the protocol, i.e., its inability to withstand multiple overlapping reconfigurations.

For the sake of clarity, the description of the RECONFIGURATION PATH protocol is split in two parts, first describing its basic operations, then continuing with the details of the management of the (un)subscriptions issued during the reconfiguration. The pseudo-code and schematic for the complete protocol are presented in Figure 11.

1) Basic Operation: This section steps through a single reconfiguration distinguishing the operations done on the head path from those made on the new link and on the tail path.

Starting the Reconfiguration. The reconfiguration process begins when the tree maintenance module invokes the action $\mathsf{substituteLinkTo}(n, RP, rId)$. As shown in Figure 11, this action, differently from the one found in INFORMED LINK ACTIVATION, provides the routing module with the entire reconfiguration path and is invoked only on the first dispatcher of the path, called the *initiator*.

The initiator removes the other end-point n of the old link from its set of neighbors and computes two sets of patterns, P_{add} and P_{del}. These contain the subscriptions that must be respectively added and removed along the head path. With reference to Figure 6, P_{add} enables the insertion of the missing ab_i subscriptions, on the path from A to C, while P_{del} enables the removal of the unnecessary subscriptions on the path from C to A. Both P_{add} and P_{del}

```
//Replace a link to dispatcher n
substituteLinkTo(n, RP, rId)
  P_add ← patterns n was subscribed to
          along old link
  P_del ← P_add
  neighbors ← neighbors − {n}
  //emulate reconfiguration message from n:
  send REC(rId, P_add, P_del, RP) to self                    [1a]

//Receive a control message from a neighbor n
recAckReceived(n, RECACK(rId))
  ignore ← ignore − {⟨rId, ∗, n⟩}                            [1d]

//Receive a flush message from a neighbor n
flushReceived(n, FLUSH(rId, RP))
  if self = first(tail(RP)) then
    for all p ∈ subTab s.t.
            p ∉ storedP_add[rId] and
            prev(RP) is not the only
            subscriber to p do
      send SUB(p) to prev(RP)                                [8]
    storedP_add[rId] ← ∅
  else
    send FLUSH to all neighbors except n                     [7]

//Unsubscription received from neighbor n
unsubscriptionReceived(n, UNSUB(p))
  if ⟨p, n⟩ ∈ subTab and
        ∄⟨∗, p, n⟩ ∈ ignore then
    processUnsubFrom(n, p)

//Receive a REC message from a neighbor n
recReceived(n, REC(rId, P_add, P_del, RP))
  if self ∈ head(RP) then
    subTab[n] ← subTab[n] − P_del
    if self ≠ last(head(RP)) then
      for all p ∈ P_add do
        ignore ← ignore ∪ {⟨rId, p, next(RP)⟩}
      P'_del ← patterns to be removed at next hop
      send REC(rId, P_add, P'_del, RP)
           to next(RP)                                       [1b]
    else
      TimedDefUnsub.addLinkTo(next(RP))                      [2]
      send REC(rId, P_add, ∅, RP)
           to next(RP)                                       [3]
    add P_add to subscriptions of next(RP)
    if self ≠ first(RP) then
      send RECACK(rId) to n                                  [1c]
  if self ∈ tail(RP) then
    if self = first(tail(RP)) then
      neighbors ← neighbors ∪ {n}
      for all p in P_add with no
              subscriber except n do
        send UNSUB(p) to n                                   [4]
      storedP_add[rId] ← P_add
    if self ≠ last(RP) then
      send REC(rId, ∅, ∅, RP) to next(RP)                    [5]
    else
      Strawman.removeLinkTo(first(RP))                       [6]
      send FLUSH(rId, RP) to self                            [7]
```

The following functions apply to a sequence of dispatchers Seq:

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

The following functions apply to the reconfiguration path RP:

head(RP)	the head path
tail(RP)	the tail path

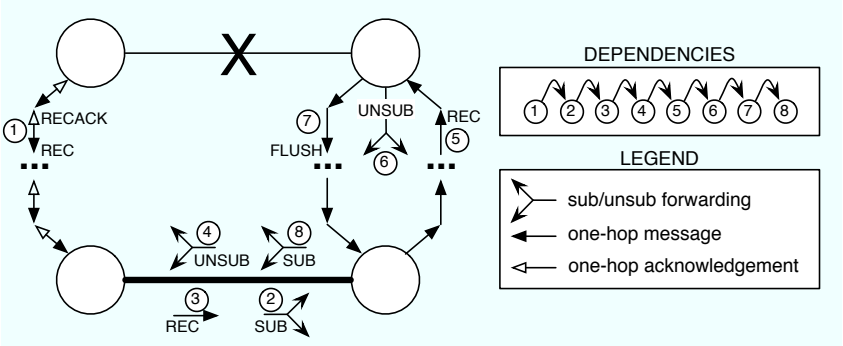


Fig. 11. RECONFIGURATION PATH pseudo-code and schematic.

are initialized by the initiator with the same patterns: those belonging to subscriptions (ab in Figure 6) previously issued by the other end-point of the vanished link.

At this point, the initiator creates a REC message containing rId, P_{add}, P_{del}, and the sequence RP, as this constitutes all the information necessary to effect the changes in the routing information along the reconfiguration path. To simplify the pseudo-code in Figure 11 and keep the processing uniform on all dispatchers, we assume that the initiator first sends the REC message to itself (step $1a$).

Reconfiguring the Head Path. The receipt of the REC message triggers the processing of the recReceived action. On the initiator, a new entry is inserted in the initiator's subscription table for each event pattern in P_{add}, as if it were a subscription coming from the next dispatcher in the reconfiguration path (E in Figure 6). This enables event forwarding towards the new link. Similarly, all the entries in P_{del} are deleted from the subscription table. In Figure 6, these actions cause respectively the insertion of ab_1 and the deletion of ab.

After reconfiguring its subscription table, the initiator propagates the REC message to its neighbor along the reconfiguration path. All along the head path, each dispatcher receiving the REC performs the same operations performed by the initiator, updating its subscription table and propagating a new REC (step $1b$). The contents of P_{add} remain the same as the REC message propagates along the head path establishing the forwarding chain to route events across the new link. The contents of P_{del}, on the other hand, are recomputed by each dispatcher (including the initiator) before propagating REC. P_{del} contains exclusively subscriptions that formerly routed events *only* towards the removed link. Therefore, if a dispatcher's subscription table contains a subscription for a pattern p to any dispatcher other than the next one on the reconfiguration path, p is not included in the P_{del} propagated to the next dispatcher.

Reconciling Subscriptions Across the New Link. The propagation of the REC message continues along the head path until it reaches the first end-point of the new link (C in Figure 6). This dispatcher behaves differently from the others along the head path as it must take the necessary steps to activate routing across the new link. In particular, it updates its subscription table as described earlier but also adds the other end-point as a new neighbor and sends it a subscription message for each pattern in its subscription table (step 2), followed by a REC message (step 3).

Completing the Reconfiguration. The other end-point of the new link processes the subscriptions

sent in step 2 normally. Consequently, these subscriptions propagate throughout the second sub-tree to enable the correct routing of events across the new link and on the tail path. Observe that since these subscriptions are generated after the removal of those on the head path and before unsubscriptions have been processed on the tail path, their propagation is naturally confined to the tail path.

After processing the subscriptions coming from the first end-point of the new link, the second end-point processes the REC message and propagates it to the next dispatcher along the tail path. Propagation of the REC message continues along the tail path (step 5) with each dispatcher simply forwarding it until it arrives at the last dispatcher. This dispatcher, which is also the second end-point of the removed link, reacts to the REC message by behaving as if it had received an unsubscription message for each subscription associated with the other end-point of the removed link. It processes these unsubscriptions and propagates them normally, completing the reconfiguration (step 6).

It is worth noting how these unsubscriptions are generated only after the subscriptions sent in step 2 have finished propagating on the tail path. This naturally limits their propagation to the reconfiguration path, removing subscriptions that were used only to route events to the first sub-tree via the removed link (e.g., the subscriptions from G to B and from D to G in Figure 6).

2) Dealing with Concurrent (Un)Subscriptions: Thus far we have ignored the details related to subscriptions and unsubscriptions issued during the reconfiguration. Different from the other protocols, RECONFIGURATION PATH demands that these be treated in a special way to avoid race conditions arising from its sequential processing.

Avoiding Race Conditions on the Head Path. The first issue arises in the head path. While the REC message flows along the head path from the initiator to the first end-point of the new link, it adds subscriptions that route events towards the next dispatcher in the head path, i.e., towards a dispatcher that has not yet received the REC message. This processing proceeds in the opposite direction w.r.t. the processing of normal subscriptions, activating a subscription before the event recipient is aware of it. To see why this can become a problem, consider two dispatchers R and S, found in this order in the head path. S is a subscriber for a pattern p. It may happen that, while R forwards the REC message to S, S simultaneously unsubscribes from p. If other subscriptions to p are present in the sub-tree of the tail path, the processing above would incorrectly remove

a subscription that, although no longer necessary to S, enables forwarding towards some other subscriber reachable through the new link. For instance, in Figure 6, this situation occurs if E unsubscribes while the REC is travelling from A to E. The unsubscription would incorrectly remove the subscription ab_1, therefore preventing events generated by the dispatchers to the left of A from correctly reaching D.

The solution we adopt requires that the REC message is explicitly acknowledged with a RECACK message, and that the sender of the REC message (i.e., R in the previous example) remembers the subscriptions added by a reconfiguration until the REC message has been acknowledged. This allows it to discern between an unsubscription that would disrupt event propagation along the reconfiguration path, and one that should instead be processed. Indeed, a dispatcher ignores the unsubscriptions received in between the forwarding of a REC and the receipt of its acknowledgement RECACK, if they correspond to subscriptions added by the reconfiguration. In Figure 11 this motivates the $ignore$ table containing the unsubscriptions that should not be processed, and the acknowledgment message RECACK, whose effect is to clear the $ignore$ table (steps $1c$ and $1d$).

Reconciling Subscriptions Across the New Link. The second issue arises because the contents of the REC message and thus the reconfiguration carried out on the head path are solely determined by the state of the initiator when the link is removed. In particular, while the REC message is on the head path, the second sub-tree is unaffected by the reconfiguration, and normal subscriptions and unsubscriptions can be issued without the possibility to reach the initiator's sub-tree.

This requires a reconciliation mechanism to remove inconsistencies generated before the two sub-trees are joined. This reconciliation is carried out by the second end-point of the new link, which compares its own subscription table with the P_{add} carried by the REC message, checking if some subscriptions added in the head path are no longer necessary because the corresponding subscribers in the second sub-tree have unsubscribed. For each subscription found in P_{add} but not in the local table, an unsubscription message is sent across the new link (step 4).

Similarly, the second end-point of the new link checks whether there are subscriptions generated in the sub-tree of the tail path during the first part of the reconfiguration, which should be added on the head path. However, this check cannot be done until the last dispatcher on the tail path receives the REC message and propagates its unsubscriptions (step 6). Therefore,

the second end-point of the new link saves the patterns in P_{add} received with the REC message into a temporary variable $storedP_{add}$, while the last dispatcher in the reconfiguration path sends a FLUSH message (step 7) after the unsubscription messages generated in step 6. By receiving this FLUSH the second end-point of the new link can determine when the reconfiguration has completed. When this happens, it compares the patterns saved in $storedP_{add}$ against its current subscription table and propagates to the first sub-tree all the subscriptions in its table that do not direct events exclusively to the other end-point of the new link and are not contained in $storedP_{add}$ (step 8).

V. EVALUATION

Section IV described in detail the behavior of our protocols. This section complements it by focusing on a numerical evaluation of their performance in several scenarios, carried out with OMNeT++ [49], a popular, open source, discrete event simulation tool.

Section V-A introduces the simulation environment and the parameters characterizing the scenarios we evaluated. Section V-B analyzes the ability of the protocols to restore the correct event routing after reconfigurations, while maintaining reasonable event delivery. Finally, Section V-C presents a detailed evaluation of the cost of dealing with reconfiguration for each of the protocols we propose.

A. Simulation Setting

In the absence of reference scenarios, we extended those we used in [37] and [18]. Clients are not modeled explicitly, as their activity affects only the dispatcher they are attached to. The parameters of our simulations and corresponding default values are shown in Table I, and briefly described below.

Events, subscriptions, and matching. Events are modeled as strings containing $\mu = 9$ random characters. Subscriptions are represented as a single character. An event matches a subscription if it contains the character specified by the subscription. Each dispatcher is allowed to subscribe to $\pi = 7$ subscriptions drawn randomly from the Π available. In most simulations we use $\Pi = 96$, limiting ourselves to the printable characters.

Publish frequency. The behavior of each dispatcher is governed by the frequency at which publish, subscribe, and unsubscribe operations are invoked by each dispatcher. The most relevant is the

Parameter	Default value
number of dispatchers	$N = 100$
dispatcher degree	$\delta = 4$
available patterns in the system	$\Pi = 96$
patterns per dispatcher	$\pi = 7$
patterns matched by each event	$\mu = 9$
density of subscribers	$\sigma_s = 0.2$
density of publishers	$\sigma_p = 1$
publish frequency at each dispatcher	$\varepsilon = 1$ pub/s
frequency of reconfiguration	$\rho = 3$ rec/s
time required to repair the tree	$t_{rep} = 0.1$ s
unsubscription timer	$T_u = 0.15$ s
subscription timer	$T_s = 0.15$ s

TABLE I

DEFAULT SIMULATION PARAMETERS. THOSE IN ITALICS REMAIN CONSTANT THROUGHOUT OUR SIMULATIONS.

publish frequency ε, which essentially determines the system load in terms of event messages that need to be routed: its impact is evaluated in Section V-C.3. In our simulations, the density of publishers is $\sigma_p = 1$, i.e., every dispatcher is a publisher. This parameter is not changed across our simulations since it affects primarily the event load, which is already controlled through the publish frequency ε.

Density of subscribers and receivers. As discussed in Section III-C, the extent to which (un)subscriptions are propagated is determined by the density of subscribers in the tree, the impact of which is analyzed in Section V-C.2.c. The choice of $\sigma_s = 0.2$ as the default value is motivated by the fact that this value causes an event to be received by approximately 10% of the dispatchers in the system—a commonly accepted "rule of thumb" for content-based systems (see, e.g., [9]). In fact, given our event model, the density of receivers for a given event can be computed as

$$\sigma_r = \sigma_s \times p = \sigma_s \left(1 - \left(\frac{\Pi - \pi}{\Pi}\right)^{\mu}\right) \tag{2}$$

where p is the probability that a given event matches at least one of a subscriber's patterns. Using the default values in Table I, Equation (2) indeed yields $\sigma_r = 0.0988 \sim 0.1$.

Network size and topology. The results we present are obtained with tree configurations consisting

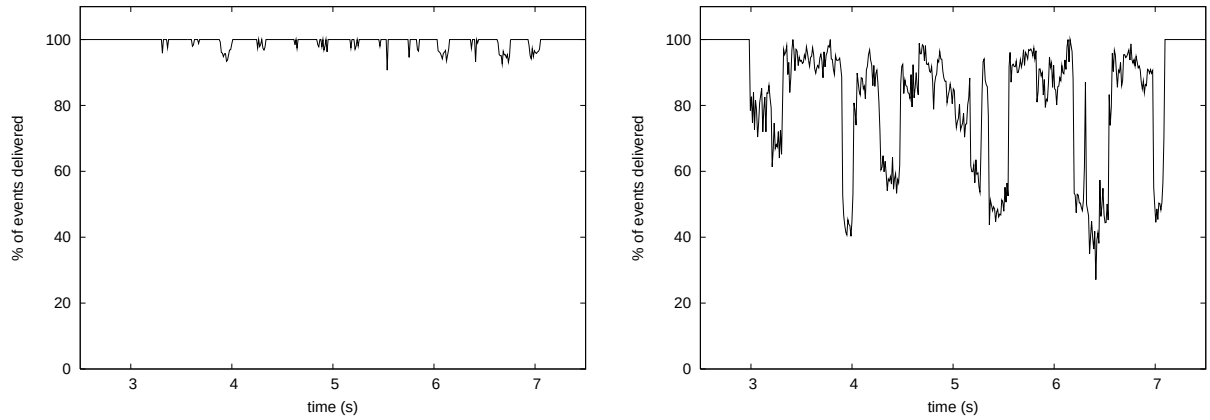
of up to 500 dispatchers, with most of our plots derived with $N = 100$. The links connecting dispatchers are assumed to behave as error-free 10 Mbit/s links. The maximum degree of the dispatchers in the network limits each dispatcher to at most $\delta = 4$ neighbors. Simulation runs with different degrees showed that the influence of this parameter is negligible. In the following, we assume that the initial configuration is a balanced tree, although in Section V-C.2.a we analyze also the case of an unbalanced initial configuration.

Tree reconfiguration. The aim of our simulations is to compare the performance of the protocols described in this paper in a situation where the dispatching tree is modified through the replacement of one link with another. The cases where the dispatching tree is partitioned into two sub-trees or two sub-trees merge are in fact treated in the same way by all the protocols we consider, and are also arguably less frequent. The selection of the links breaking or appearing is done randomly. However, the same random sequences were applied to all the protocols in order to obtain consistent results. To retain some degree of control about when a reconfiguration occurs, we assume that each broken link is replaced by a new one after $t_{rep} = 0.1s$.

Each simulation represents an interval of eight seconds. During the first three seconds, dispatchers operate normally, generating subscriptions, unsubscriptions and events in the absence of topological reconfigurations. These occur in the interval between $3s$ and $7s$, at a regular frequency determined by the parameter ρ, whose impact is assessed in Section V-C.2.d. The last second is used to allow reconfigurations to complete.

Timers. Some of our protocols make use of timers to coordinate their actions. The choice of the best timer values depends on the specific scenario being considered. In the scenario we analyzed, however, we set both the subscription and the unsubscription timers to $0.15s$, as these values yield average performance. An analysis of the impact of timer values is provided in Sections V-C.2.e and V-C.3.b.

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. The same set of seeds is used for all the protocols evaluated in each configuration. Instead, in the case of unbalanced tree configurations the initial tree is random and different for each run.



(a) $\rho = 3$ reconfiguration/s (b) $\rho = 30$ reconfiguration/s

Fig. 12. Event delivery for the TIMED DEFERRED UNSUBSCRIPTION protocol.

B. Event Delivery

The first property we need to evaluate in our protocols is their ability to restore correct event routes regardless of the temporary disruption caused by reconfigurations. If the protocols behave correctly, the percentage of events delivered should drop temporarily as a consequence of reconfiguration, and then go back to 100%. This is exactly the case in Figure 12, where we show the results obtained by the TIMED DEFERRED UNSUBSCRIPTION protocol under different reconfiguration rates. We report only the results obtained with one of the protocols, as simulation of the others did not evidence significant differences.

The measurements were performed by relying on a subset of the dispatchers belonging to a *stable core*. Core dispatchers are prevented from issuing (un)subscriptions after a given time threshold, set to $2s$ in our tests. The presence of the stable core focuses our measurements on the events lost as a result of reconfigurations, eliminating events missed during the propagation of new subscriptions. Nevertheless, only the stable core is subject to this limitation: as a result, the protocols are validated not only against the reconfiguration coming from changes in the topology, but also against the reconfiguration of routing information determined by the (un)subscriptions coming from dispatchers not in the core.

The plots of Figure 12 are based on a configuration with $N = 100$ 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, and a high event load of $\varepsilon = 50$ publish/s is assumed. The reconfiguration rate ρ in Figure 12(a) allows each reconfiguration to complete before the next one starts. In this case, event delivery is only marginally affected by reconfiguration. Instead, the higher rate of Figure 12(b) leads to a situation where reconfigurations overlap in time and space, therefore negatively affecting event delivery.

Figure 12 shows how, independently of the reconfiguration scenario, our protocols always restore correct routes. Indeed, event delivery goes back to *exactly* 100% after the network topology stabilizes at $t = 7$.

C. Overhead

The simulation results for event delivery indicate that our protocols correctly restore event routes in the presence of topological reconfigurations, but do not provide insights about the efficiency of the process. Here, we evaluate this aspect by focusing on the communication overhead, while Section V-C.5 analyzes also the number of nodes involved in (i.e., triggering some processing as a consequence of) a reconfiguration, as an indirect measure of the computational overhead induced in the dispatching network.

In the plots we present in this section, the main quantity under evaluation is the (average) cost of a single reconfiguration, computed by taking the number of overhead messages generated during the simulation run and dividing it by the number of reconfigurations that occurred. This quantity represents the most basic "building block" necessary to assess the behavior of our protocols, and in the rest of this section we show how it varies according to changes in the simulation parameters of Table I. Each plot reports the original data points together with their Bezier interpolation, to help visualize trends. Also, note that the overhead is measured by making all dispatchers part of the stable core. This way, the only (un)subscription messages exchanged in the system are those caused by reconfiguration.

Before delving into the analysis, however, we detail further how we modeled the various overhead components.

1) Modeling the Cost of Publish-Subscribe and Control Messages: Throughout the analysis, the communication overhead is computed as the number of messages that the protocols generate to restore correct event routing in the presence of reconfigurations. More precisely, the overhead is the sum of: *i)* the (un)subscription messages exchanged because of reconfiguration; *ii)* the control

Message	Weight
SUB	1
UNSUB	1
EVENT	1
FLUSH	0.1
ACTIVATE	#patterns
REC	#patterns
RECACK	0.1

TABLE II

MODELING THE DIFFERENT COSTS OF PUBLISH-SUBSCRIBE MESSAGES AND CONTROL MESSAGES.

messages of the protocols that manage the reconfiguration process; and *iii)* the event messages misrouted along obsolete subscription paths and therefore reaching uninterested dispatchers. Obviously, based on what we presented in Section IV, not all of these overhead components are necessarily present in all of the protocols.

The overhead generated by a message depends both on the number of hops it travels and on its size. Nevertheless, the actual size of event and (un)subscription messages is ultimately determined by the application, while the size of control messages is determined by the middleware implementation. Simply counting the number of messages generated is misleading, since the difference in size among these messages is significant. For instance, a FLUSH message is likely to be very small since it only carries an identifier, while a REC message contains a set of patterns and therefore its cost is roughly equivalent to the sum of the sizes of the messages issued to subscribe to those patterns.

In our evaluation we assigned different weights to the various messages, to account for their different sizes. We assume that a subscription, unsubscription, or event message have the same size c, analogously to other researchers in the field (e.g., [47]). This value, which we leave undefined as it depends on the implementation, is used as the base to derive the cost of control messages as $w \times c$, where w is a weight associated to the message type, according to Table II. We used $w = 1$ for the aforementioned standard publish-subscribe messages, $w = 0.1$ for control messages that do not carry patterns, and a value of w equal to the number of patterns contained in the message for the remaining ones. The *normalized cost* generated by each message is then

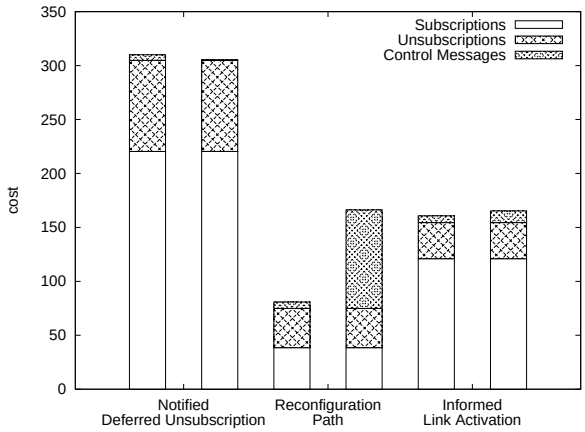


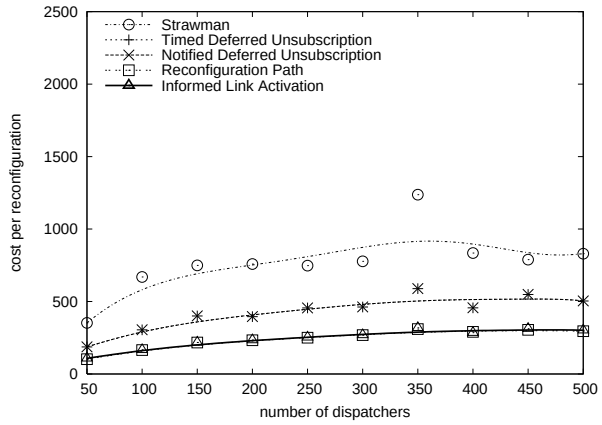
Fig. 13. Absolute number of messages vs. normalized cost, in a scenario defined by the parameters in Table I and by $\varepsilon = 0$. For each protocol, the left bar shows the absolute number of messages exchanged, the right bar the normalized cost obtained using the weights in Table II. Only protocols with control messages are shown.

computed by multiplying its weight by the number of hops it travels, and dividing it by c. Note that each out-of-band message is considered to travel for one hop, since we assume that TCP or some other point-to-point communication protocol is available between dispatchers.

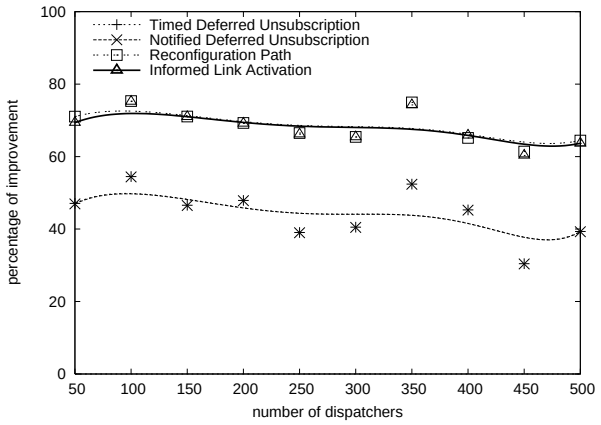
The bias introduced by this modeling of overhead can be appreciated by looking at Figure 13, which reports simulation results obtained in the reference scenario defined by Table I. The figure shows, for each protocol containing control messages, the absolute number of overhead messages exchanged on the left-hand side and the normalized cost on the right-hand side. It is worth noting how our modeling choice is a conservative one, in that it actually lowers the performance figures of our protocols. In fact, while the impact of FLUSH and RECACK messages is in any case negligible when compared to that of (un)subscriptions, the absolute number of REC and ACTIVATE messages is instead much lower than their normalized cost (e.g., going from 3.44 to 90.9 for REC messages).

All the simulation plots we illustrate in the remainder of this section show the normalized cost, unless otherwise stated.

2) Evaluating the Cost of Reconfiguring Subscription Tables: Our evaluation begins by investigating the cost of restoring the consistency of subscription tables after topological reconfigurations. We analyze this major component of overhead in isolation, i.e., when no events are being published in the system ($\varepsilon = 0$). Therefore, overhead is solely determined by (un)subscriptions

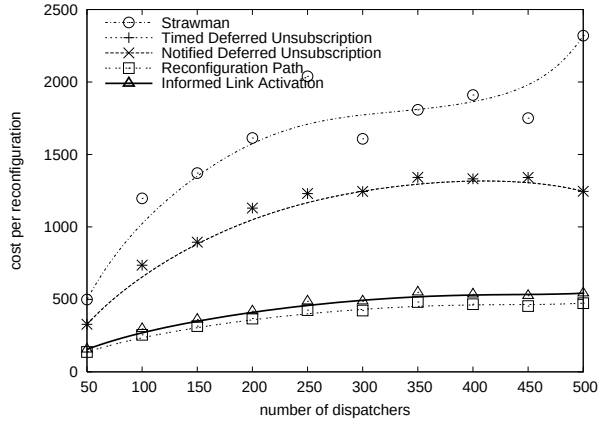


(a) Unit cost per reconfiguration.

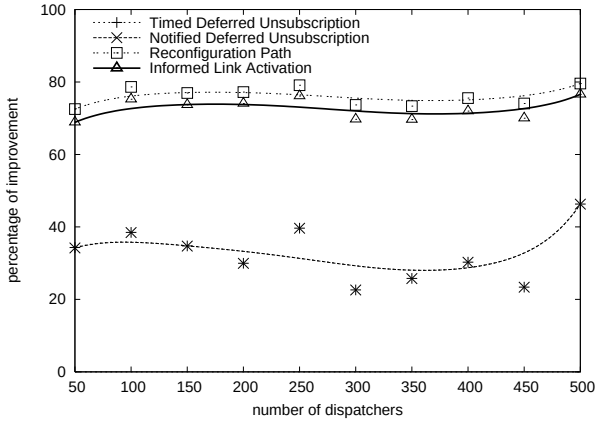


(b) Percentage improvement over STRAWMAN.

Fig. 14. Cost of reconfiguration vs. system scale.



(a) Unit cost per reconfiguration.



(b) Percentage improvement over STRAWMAN.

Fig. 15. Cost of reconfiguration vs. system scale, with unbalanced initial tree configurations.

and control messages. The impact of misrouted events, which is nonetheless negligible in the reference scenario, is analyzed in Section V-C.3 and following.

a) System Scale: We begin by analyzing the performance of our protocols against the system scale, by ranging the network size N from 50 to 500 dispatchers and keeping the other parameters of Table I unaltered. Note how this really represents an increase in system scale and not just in the network size. Indeed, an increase of N causes a corresponding increase in the number of subscribers, which is defined in terms of the density σ_s. Moreover, we also increase the number of available patterns to $\Pi = 200$ to account for the increased scale. The impact of

this latter parameter is analyzed in more detail in Section V-C.2.b.

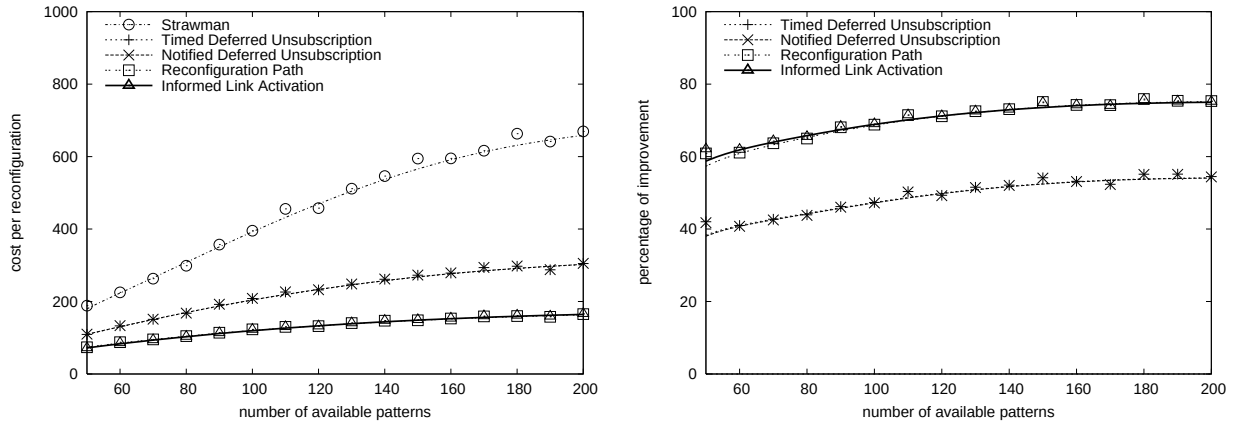
Figure 14 shows our simulation results. Figure 14(a) shows the average cost of a reconfiguration for each protocol, including STRAWMAN. The plot evidences how this cost increases along with the size of the network, as the distance between the subscribers on the pattern tree becomes longer. Nevertheless, as can be appreciated from the percentage improvement over STRAWMAN plotted in Figure 14(b), all of our protocols consistently and remarkably outperform the STRAWMAN protocol, reducing overhead by up to 75%. Also, it can be noted how the solutions based on deferred unsubscriptions are less effective than the others, with a gap in performance of about 20%.

The plots in Figure 14 also evidence how the performance of the two variants of DEFERRED UNSUBSCRIPTION is virtually indistinguishable. This is not surprising, as they have the same fundamental behavior, and differ only in the mechanism used to trigger the unsubscriptions previously deferred. Moreover, since the FLUSH messages used by NOTIFIED DEFERRED UNSUBSCRIPTION are small in size and few in number, as discussed in Section V-C.1, their impact on overhead is negligible. Analogously, the similarity between INFORMED LINK ACTIVATION and RECONFIGURATION PATH can be explained by observing that both limit the scope of the reconfiguration to the reconfiguration path. The similarity between the protocols is also a result of the specific scenario we considered, one without event load and with non-overlapping reconfigurations. Later on in our analysis, we show that when these dimensions are considered the various protocols exhibit different performance and tradeoffs.

The results in Figure 14 were obtained by starting each simulation with a balanced tree topology. This choice allows us to remove an additional source of randomness from our results, and for this reason we retain it throughout this section. Nevertheless, here we evaluate the impact of the initial tree configuration. In general, a tree with a random topology has a larger diameter (depth) than the corresponding balanced tree. As a result, we expect a random initial configuration[4] to amplify the differences among the various protocols because the overhead messages, on average, travel longer than in the balanced case.

This is confirmed by comparing Figure 15(a) against Figure 14(a). Also, comparison of Figure 15(b) and 14(b) shows that RECONFIGURATION PATH and INFORMED LINK ACTIVATION

[4]The unbalanced configuration is random, but the maximum number of neighbors is still fixed, $\delta = 4$ in our simulations.



(a) Unit cost per reconfiguration.

(b) Percentage improvement over STRAWMAN.

Fig. 16. Impact of the number Π of available patterns on the reconfiguration overhead.

improve an additional 10% against STRAWMAN with respect to the balanced case, while the relative performance of the DEFERRED UNSUBSCRIPTION protocols drops by about the same quantity. The explanation is straightforward: the STRAWMAN and the DEFERRED UNSUBSCRIPTION protocols are affected by the increase in the distance between dispatchers on the pattern tree causing overhead messages to travel longer. Instead, INFORMED LINK ACTIVATION and RECONFIGURATION PATH are less affected by the increased diameter of the network. The reason is that in the absence of concurrent subscriptions and unsubscriptions both protocols manage to confine overhead messages to the reconfiguration path. INFORMED LINK ACTIVATION exploits the information in the ACTIVATE messages to avoid the propagation of unnecessary (un)subscriptions outside the reconfiguration path, while RECONFIGURATION PATH achieves even better performance by virtue of the sequential process established by the REC along the reconfiguration path.

Finally, it is worth noting that since the overhead in this unbalanced scenario is higher for all the protocols, the absolute savings provided by our protocols over STRAWMAN are larger.

b) Number of Available Event Patterns: In our simulations, each subscriber holds π subscriptions, each for a pattern randomly drawn from the Π patterns available in the system. In this section, we analyze how changes in Π affect the performance of the various protocols. In real systems, as the scale of the system increases Π increases as well (albeit not necessarily at the same rate) since there are new subscribers with unique subscriptions. The charts in Figure 16,

derived for $N = 100$, show that our protocols improve w.r.t. STRAWMAN as the number of available patterns increases, therefore confirming that our choice of a default $\Pi = 96$ patterns is actually conservative.

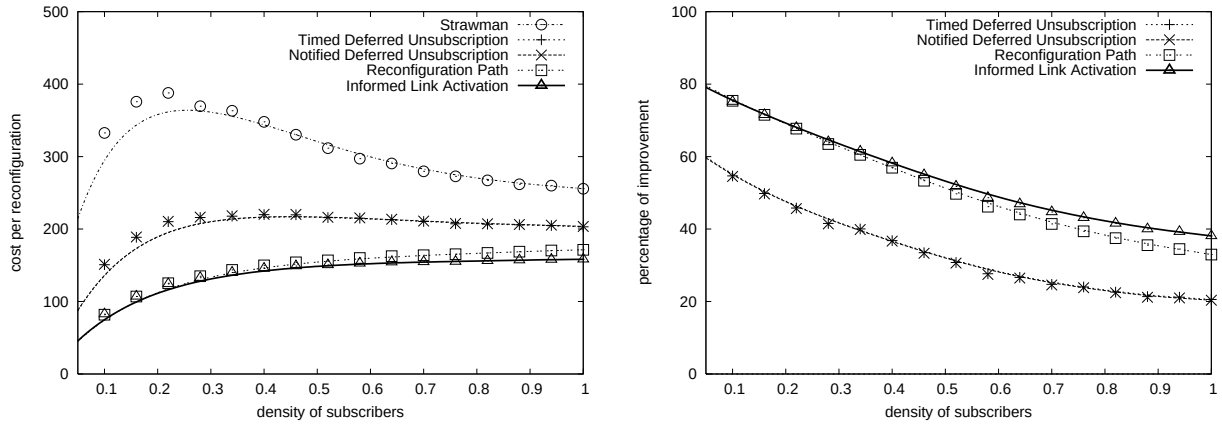
Indeed, the STRAWMAN protocol is negatively affected by a larger number of available patterns. An increase in the number of patterns results in a lower density of subscribers per pattern, therefore increasing the distance travelled by subscriptions and unsubscriptions to join the pattern tree. Different from STRAWMAN, our protocols manage to limit this distance by either deferring unsubscriptions (therefore preserving the initial density of subscribers) or forcing reconfiguration messages to remain on the reconfiguration path. Figure 16(b) confirms this intuition by showing that the improvement of all our optimized protocols increases with the number of available patterns.

The arguments we put forth in this section justify our choice of $\Pi = 200$ in Section V-C.2.a, to accommodate for the increased scale. Moreover, they also explain why the improvement curve in Figure 14(b) exhibits a small decrease when the size of the dispatching network increases. This trend is a consequence of our choice of a fixed number of available patterns. When the number of nodes, and thus the density of subscribers, increases the tree becomes dense of subscriptions, since the π subscriptions for each subscriber are drawn from the same, fixed set of Π patterns. This reduces the total number of hops traveled by subscription and unsubscription messages, and correspondingly reduces the gap between STRAWMAN and the optimized protocols. In a true content-based system this "saturation" phenomenon is unlikely to occur, since an increase in scale is usually mirrored by some increase in the number of available patterns.

In the rest of this section we focus on a network of $N = 100$ dispatchers and retain the default value $\Pi = 96$, unless otherwise stated.

c) Density of Subscribers: The average cost of a reconfiguration clearly depends on the density of subscribers in the dispatching network, as we discussed in Section III-C. The higher the density of subscribers the shorter the distance travelled by (un)subscription messages caused by reconfiguration, and consequently the smaller the improvement achieved by our protocols. Figure 17 confirms this intuition by showing the results of simulations in the reference scenario of Table I, changing the density of subscribers in the range $3\% \leq \sigma_s \leq 100\%$, which according to Equation (2) yields a number of receivers per event in the range $1.48\% \leq \sigma_r \leq 49.40\%$.

It is worth making two additional observations about Figure 17. First, even with a tree



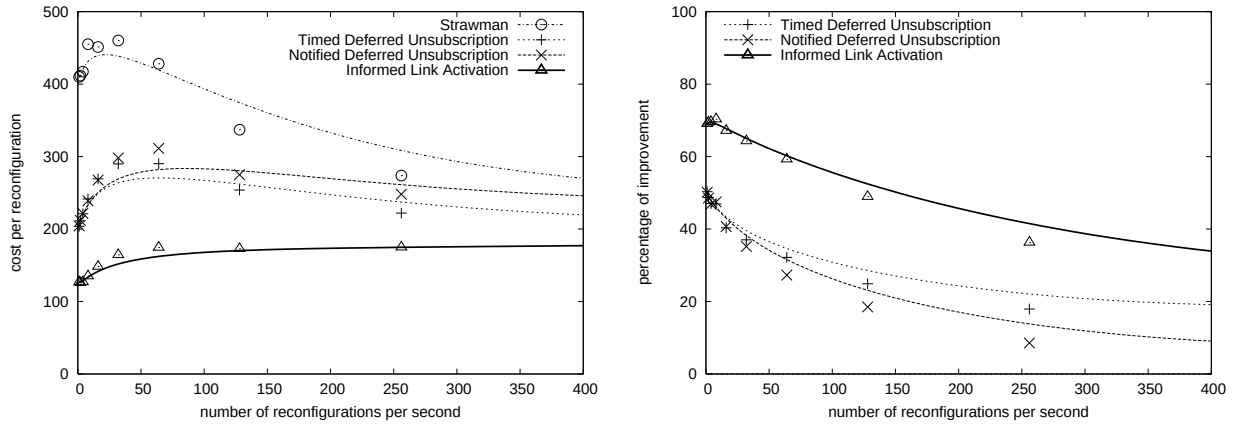
(a) Unit cost per reconfiguration.

(b) Percentage improvement over STRAWMAN.

Fig. 17. Impact of subscriber density σ_s on reconfiguration overhead.

where all dispatchers are also subscribers, our protocols are still able to significantly improve over STRAWMAN, from the 20% improvement achieved by the DEFERRED UNSUBSCRIPTION protocols to the 40% achieved by INFORMED LINK ACTIVATION. Second, the higher the density of subscribers the less the scenario faithfully represents a content-based system. In content-based systems, patterns are usually highly selective, and the number of receivers per event is usually assumed to be reasonably low (about 10% for $\sigma_s = 0.2$, as discussed in Section V-A), as this is one of the aspects differentiating content-based communication from multicast and broadcast communication. Instead, here σ_r is well beyond these commonly assumed values. Additionally, while a high subscriber density conflicts with the sheer notion of content-based publish-subscribe, very small values of σ_s (and therefore σ_r) are meaningful in some application scenarios where a small number of devices is responsible for collecting data published by a large number of data sources. For instance, this situation is typical of monitoring and sensing applications, such as those recently made popular by wireless sensor networks [1]. Interestingly, in these applications reducing the communication overhead induced by the monitoring infrastructure is of paramount importance.

d) Frequency of Reconfiguration: Thus far, we have considered a reconfiguration rate $\rho = 3$ rec/s. In our reference scenario with $N = 100$ dispatchers and a time to reconnect the tree $t_{rep} = 0.1$ s, this leads to reconfigurations that complete before a new one starts, and therefore can be considered as occurring in isolation. Higher reconfiguration rates, instead, are likely to



(a) Unit cost per reconfiguration.

(b) Percentage improvement.

Fig. 18. Impact of reconfiguration rate on reconfiguration overhead.

generate reconfigurations that overlap not only in time (i.e., occurring in parallel) but also in space (i.e., involving a common portion of the dispatching tree). Our choice for the default value of ρ was motivated by the desire to reduce interference from different phenomena and to enable the evaluation of the RECONFIGURATION PATH protocol, which does not tolerate overlapping reconfigurations. Here, we analyze the impact of a change in this parameter, and we do so without considering the RECONFIGURATION PATH protocol, due to its limitations.

Figure 18 reports the simulation results obtained by changing the reconfiguration rate in the range $1 \leq \rho \leq 400$ rec/s. With $N = 100$, the upper bound leads to each link experiencing about 4 breakages per second. Once more, this value is not necessarily meant to mirror a realistic reconfiguration rate, rather to elicit the behavior of our protocols in extreme conditions[5].

The most prominent phenomenon evidenced by the charts is the fact that the average cost of a reconfiguration does not remain constant with the reconfiguration rate. Therefore, reconfigurations cannot be considered independent: multiple reconfigurations occurring in parallel do interfere with each other. At reasonable reconfiguration rates an increase of ρ corresponds to an increase in the average cost of a reconfiguration. This is caused by reconfigurations occurring in parallel and "partially undoing" each other, e.g., removing subscriptions that are immediately restored by a new reconfiguration, or vice versa. The more a protocol relies on standard propaga-

[5]We actually experimented with even higher rates, without finding significant differences beyond 400 rec/s.

tion of (un)subscriptions the more evident is the phenomenon. Indeed, STRAWMAN experiences the biggest increase, while INFORMED LINK ACTIVATION experiences only a limited, albeit steady, increase. On the other hand, after a given point—different for all protocols—an increase of the reconfiguration rate causes a decrease in the average cost of a reconfiguration. The reason is that the dispatching tree becomes so disrupted that the reconfiguration cost becomes more and more dominated by the exchange of subscriptions (or unsubscriptions) occurring when a link appears (or vanishes). Indeed, as the reconfiguration rate increases the differences between the various protocols become less and less significant, although our solutions always improve over STRAWMAN. Also, note how the performance of the NOTIFIED DEFERRED UNSUBSCRIPTION protocol becomes worse than TIMED DEFERRED UNSUBSCRIPTION as the reconfiguration rate increases. This is caused by the fact that former triggers unsubscriptions earlier than the latter. Leaving stale subscriptions in place for longer enables TIMED DEFERRED UNSUBSCRIPTION to benefit from the case where the subscription is going to be re-established by another, concurrent reconfiguration, altogether removing the need for the unsubscription.

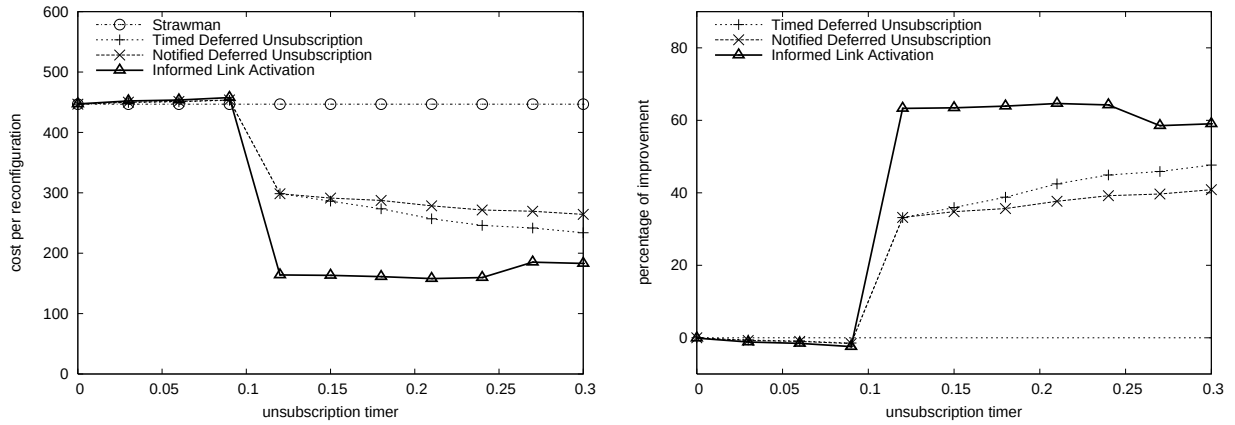
As we mentioned, however, these behaviors are found only in extreme reconfiguration scenarios that are unlikely to occur in practice. In real world settings, the reconfiguration rate is likely to fall to the very left of the charts in Figure 18, where the benefits of our protocols are larger.

e) Timers: Some of our protocols make use of the timers T_u and T_s to synchronize the propagation of subscriptions with the removal of stale ones. In this section, we analyze their effect on the reconfiguration of routes in the subscription tables, therefore still assuming $\varepsilon = 0$. In Section V-C.3.b we instead evaluate their impact on misrouted events when the publish rate is $\varepsilon \neq 0$.

Unsubscription Timer. Figure 19 illustrates the effects of variations to T_u. To amplify these effects, which would otherwise be negligible in our reference scenario, we increased the reconfiguration rate by an order of magnitude, bringing it to $\rho = 30$ rec/s.

Figure 19(a) evidences the presence of a marked discontinuity around $T_u = t_{rep}$, as expected[6]. This discontinuity is *always* present regardless of the value of ρ. If the timer value is too small, our protocols tend to behave in the same way as STRAWMAN because unsubscriptions

[6]More precisely, the discontinuity occurs around $T_u = t_{rep} + t_{prop}$. However, since the unsubscription propagation time t_{prop} is negligible in our simulations, we do not consider it.



(a) Unit cost per reconfiguration.

(b) Percentage improvement over STRAWMAN.

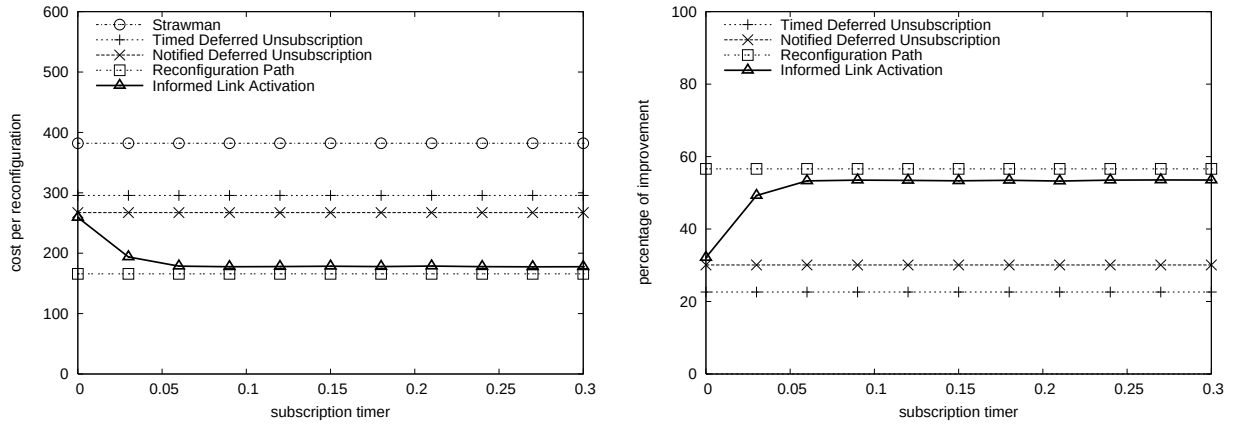
Fig. 19. Impact of the unsubscription timer on reconfiguration overhead, at $\rho = 30$ rec/s. The STRAWMAN protocol is shown as a term of comparison.

are triggered too early. Therefore, the timer should be set large enough to allow the propagation of subscriptions to complete before the propagation of unsubscriptions starts.

Interestingly, the TIMED DEFERRED UNSUBSCRIPTION protocol is positively affected by large timer values as they allow the protocol to reduce the number of unnecessary (un)subscriptions and hence to reduce the overhead. The phenomenon is due to the same effect observed in Section V-C.2.d: the removal of a stale route may become unnecessary if another reconfiguration requires establishing the same subscription, and increasing the timer value increases the chance that this situation occurs. NOTIFIED DEFERRED UNSUBSCRIPTION experiences a similar, albeit smaller, improvement, since the influence of T_u is diminished by the presence of the notification mechanism, which is usually triggered before T_u expires.

As for INFORMED LINK ACTIVATION, a discontinuity around $T_u = t_{rep}$ is present as in the other two protocols, along with a second one around $T_u < T_s + t_{rep}$. The second discontinuity is due to the interaction between the two timers T_u and T_s, according to Equation (1). If the constraint $T_u < T_s + t_{rep}$ we introduced in Section IV-B does not hold (i.e., for $T_u > 0.25$ s in our charts) the overhead of INFORMED LINK ACTIVATION increases, since the unnecessary subscriptions that have been "held" at the end-points of the new link are released before unsubscriptions have finished propagating, causing unnecessary overhead.

Therefore, we can conclude that for what concerns the cost of reconfiguring subscription



(a) Unit cost per reconfiguration.

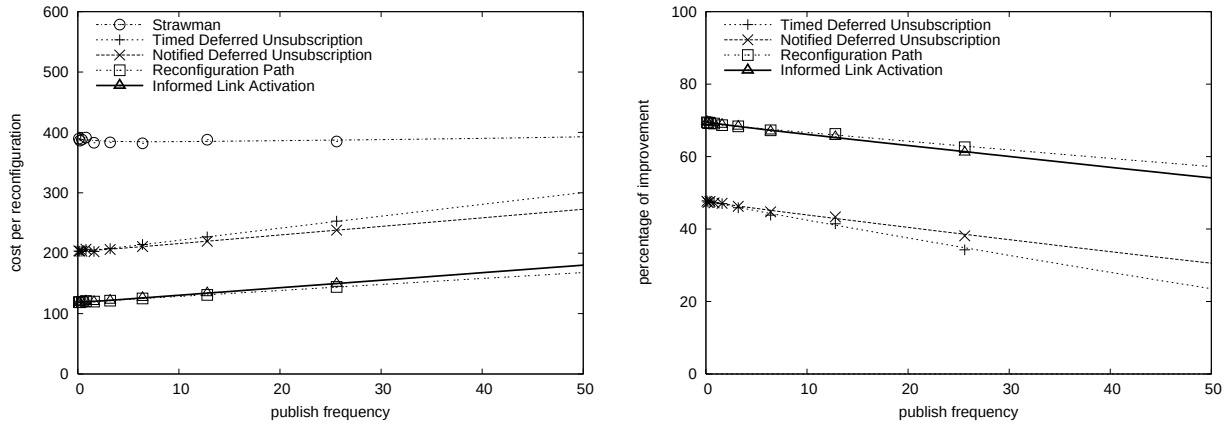
(b) Percentage improvement over STRAWMAN.

Fig. 20. Impact of the subscription timer on reconfiguration overhead. The STRAWMAN, DEFERRED UNSUBSCRIPTION and RECONFIGURATION PATH protocols are shown as terms of comparison.

tables, as long as $T_u > t_{rep}$ (or $t_{rep} < T_u < T_s + t_{rep}$ for INFORMED LINK ACTIVATION) our protocols always improve over STRAWMAN, and do not show significant dependency on the value of the unsubscription timer. Nevertheless, the tradeoffs may be different in the presence of event traffic, as we discuss in Section V-C.3.b.

Subscription Timer. Similar considerations hold for the subscription timer T_s employed only by the INFORMED LINK ACTIVATION protocol. This second timer is used to delay the propagation of those subscriptions that were used only to route events across the broken link when the break occurred. The timer should therefore be set to a large enough value ($T_s > T_u - t_{rep}$, as discussed in Section IV-B) to allow the unsubscriptions to propagate from the broken link to its replacement. Too small of a value allows for some unnecessary subscriptions to propagate, therefore making the INFORMED LINK ACTIVATION protocol behave similarly to the DEFERRED UNSUBSCRIPTION protocols. These considerations are mirrored in the charts in Figure 20, derived with the default reconfiguration rate $\rho = 3$ rec/s.

Differently from the unsubscription timer T_u, a large value for T_s is always beneficial, or in the worst case irrelevant, to the overhead. Nonetheless, it can cause a significant decrease in the event delivery, since setting the timer too large delays the subscriptions issued during a reconfiguration until T_s expires, and therefore affects the delivery of events towards the corresponding subscribers.



(a) Unit cost per reconfiguration.

(b) Percentage improvement over STRAWMAN.

Fig. 21. Impact of publish frequency on reconfiguration overhead.

3) Evaluating the Impact of Misrouted Events: We now investigate the performance of our protocols when event traffic is injected in the system, that is, $\varepsilon \neq 0$. In this case, inconsistencies in the subscription tables lead to misrouted events, which increase the overhead.

a) Publish Frequency: At the publish frequency of $\varepsilon = 1$ pub/s we selected for our reference scenario, the impact of misrouted events is negligible. Nevertheless, their impact becomes significant at higher publish frequencies. Here, we analyze the performance of our protocols with a publish frequency varying in the range $0.1 \leq \varepsilon \leq 51.2$ pub/s, where the distance among data points follows a geometric progression. To put these values in context, the publish rate of applications dominated by human interaction, such as collaborative work in mobile environments, is arguably comparable to—and, more likely, much lower than—1 publish/s *per dispatcher*, which is in fact the default value we chose in Table I. This is especially true in applications where the most natural design involves co-locating each client with a dedicated dispatcher on a network host, as in peer-to-peer or mobile ad hoc networks [27]. The upper bound of more than 50 pub/s is instead almost equivalent to a streaming application. Therefore, the high publish frequency in the chart should be regarded mostly as a way to evaluate our protocols in an extreme, and almost unrealistic situation.

Figure 21 shows the reconfiguration overhead, while Figure 22 reports the absolute number of misrouted events per reconfiguration for each protocol. A different view is provided in Figure 23, which exemplifies the relative impact of misrouted events w.r.t. the other components

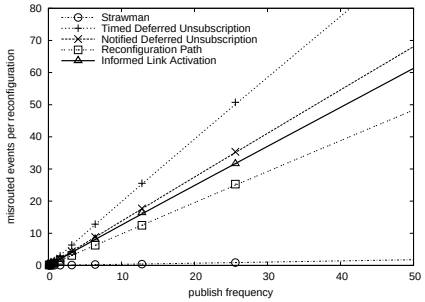


Fig. 22. Absolute number of misrouted events per reconfiguration.

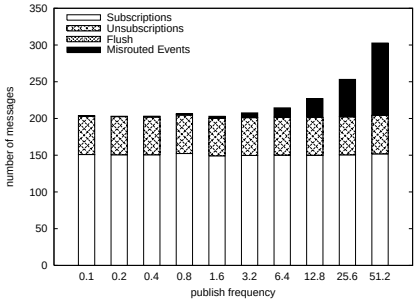
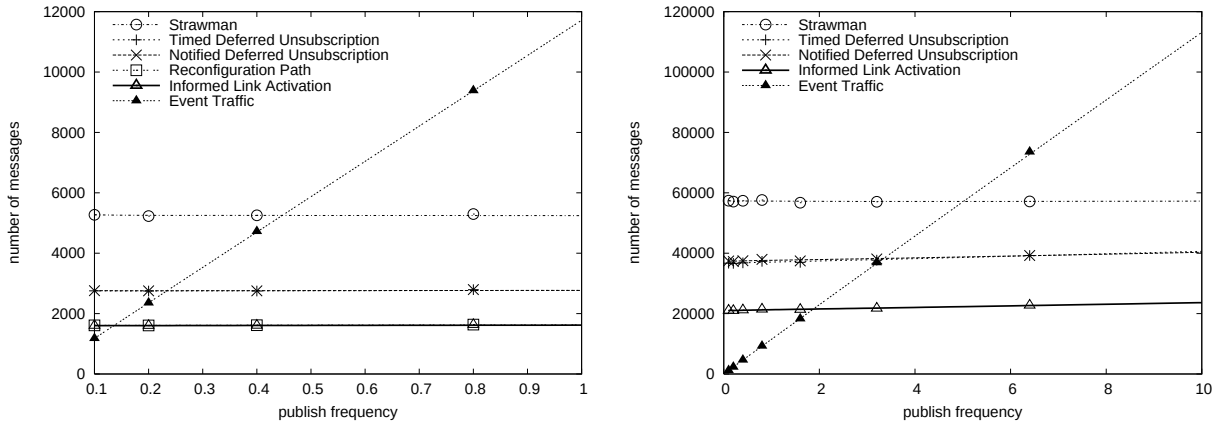


Fig. 23. Misrouted events vs. the other overhead components in TIMED DEFERRED UNSUBSCRIPTION.

as the publish frequency increases. Interestingly, STRAWMAN generates a negligible number of misrouted events even at a high publish rate, as it removes immediately stale routes and does not propagate unnecessary subscriptions. Instead, all of our protocols suffer from the presence of misrouted events, although the performance drop they induce is somewhat limited. The DEFERRED UNSUBSCRIPTION protocols perform the worst since they base their operation on propagating subscriptions (possibly including some unnecessary ones) before unsubscriptions. This generates a larger number of misrouted events with respect to the other protocols because stale routes remain active for longer periods. However, NOTIFIED DEFERRED UNSUBSCRIPTION performs better than TIMED DEFERRED UNSUBSCRIPTION, since the notification mechanism enables the triggering of unsubscriptions without waiting for the expiration of the timer T_u. The other two protocols perform better as they allow inconsistent routing tables only on the reconfiguration path; INFORMED LINK ACTIVATION performs a little worse than RECONFIGURATION PATH, as it waits for longer before removing stale routes.

As we mentioned, however, we push the publish frequency to such an unrealistic high event load mostly to stress the performance of our protocols and elicit the various constituents of overhead. Provided that an application with such a high event load exists, the optimization provided by any protocol is likely to be dwarfed by the sheer number of published events. Figure 24 plots on the same chart the "goodput" generated by events delivered to the intended receivers (i.e., the total number of events minus the misrouted ones) and the traffic generated by reconfiguration. Both metrics are plotted against the publish frequency ε, in the reference



(a) $\rho = 3$ reconfiguration/s

(b) $\rho = 30$ reconfiguration/s

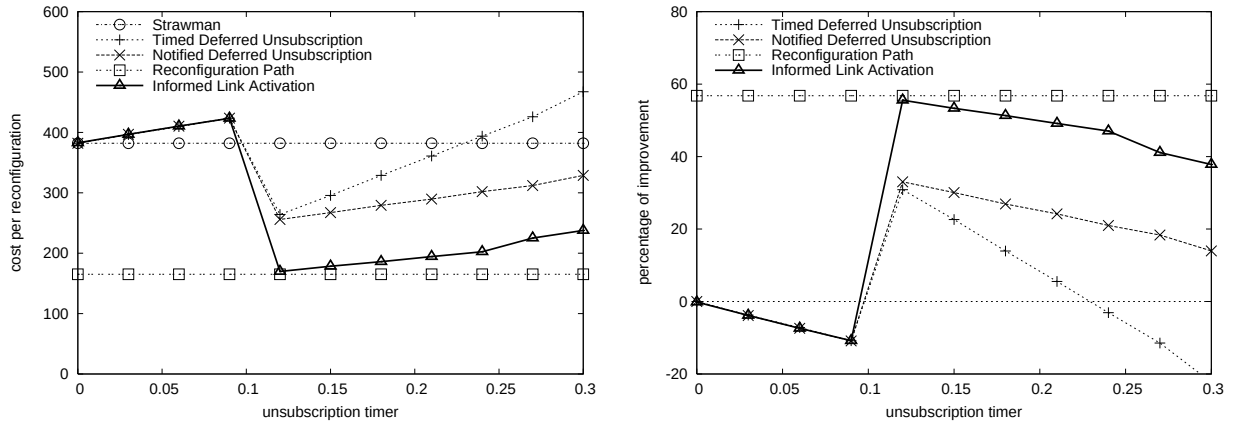
Fig. 24. "Good" event traffic vs. reconfiguration overhead.

scenario defined by Table I. Figure 24(a) shows that at $\varepsilon = 1$ pub/s the overhead generated by STRAWMAN is in a ratio of about 1:2 with the traffic generated by events, while INFORMED LINK ACTIVATION brings this ratio down to 1:10. Smaller values of ε show how our protocols yield even more remarkable improvements. In addition, it is worth observing that the overhead depends not only on ε but also on ρ: as shown in Figure 24(b), a higher reconfiguration rate shifts the overhead curves higher, therefore changing significantly the tradeoffs, making our optimizations more relevant.

b) Timers: The subscription timer T_s may prevent events from reaching subscribers by delaying the establishment of the corresponding routes, but it bears no effect on misrouted events, and therefore it is not considered here.

On the other hand, the presence of misrouted events may significantly affect the considerations we made in Section V-C.2.e for the unsubscription timer, as a large value for T_u may misroute events through stale routes towards areas of the network with no subscribers. As we showed in Section V-C.3.a, however, a very small number of misrouted events is generated at $\varepsilon = 1$ pub/s in our reference scenario. As a consequence, the differences obtained by changing the value of the unsubscription timer T_u are minimal. Therefore, once more, for the sake of eliciting the behavior of our protocols by amplifying the effects of timers, we use a very high publish frequency $\varepsilon = 50$ pub/s.

Figure 25 confirms the above reasoning. As in Figure 19, there is a discontinuity around



(a) Unit cost per reconfiguration.

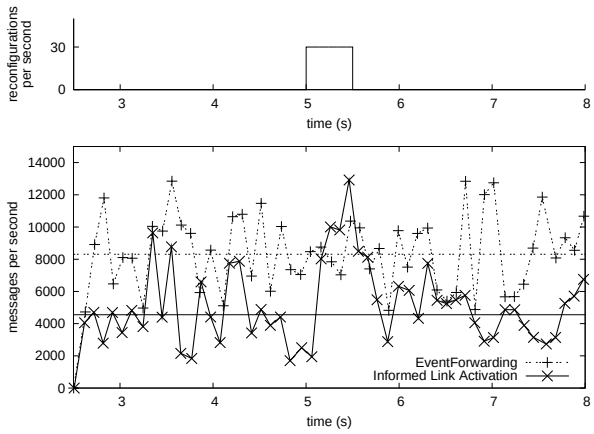
(b) Percentage improvement over STRAWMAN.

Fig. 25. Impact of the unsubscription timer on reconfiguration overhead. The STRAWMAN, and RECONFIGURATION PATH protocols are shown as terms of comparison.

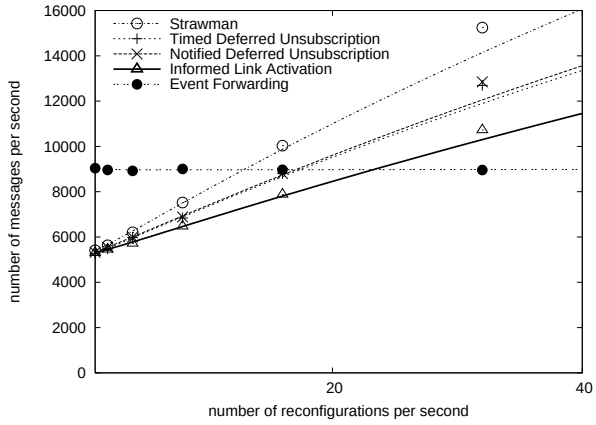
$T_u = t_{rep}$. Differently from Figure 19, however, before and after this value the performance penalty the protocols incur is determined by misrouted events flowing along broken ($T_u < t_{rep}$) or stale ($T_u > t_{rep}$) routes. Moreover, at the high publish rate we chose, the performance drop is evident even with the default reconfiguration rate of $\rho = 3$ rec/s. Clearly, the TIMED DEFERRED UNSUBSCRIPTION protocol is the most negatively affected, as its behavior depends heavily on the value of T_u. NOTIFIED DEFERRED UNSUBSCRIPTION and INFORMED LINK ACTIVATION are less affected, since they resort to T_u only when their main mechanism to trigger unsubscriptions (i.e., FLUSH messages) fails.

4) A Note About Event Forwarding: One could claim that the overhead caused by the reconciliation of subscription tables could be eliminated by avoiding keeping routes altogether, and therefore resorting to the event forwarding strategy outlined in Section II. If events are rarely published, or more generally subscriptions are changed much more frequently than events are published, then clearly event forwarding is preferable. However, this is a well-known tradeoff that exists regardless of reconfiguration, as discussed in Section II. The question we investigate here, instead, is whether reconfiguration narrows the gap between the event forwarding and subscription forwarding strategies, and therefore makes the former preferable.

Figure 26(a) crisply illustrates the tradeoffs at stake by showing the total traffic generated by event forwarding and subscription forwarding, the latter extended with our INFORMED LINK



(a) Traffic vs. time. A reconfiguration burst occurs in the interval $5s \leq t \leq 5.5s$ at $\rho = 30$ rec/s. The horizontal lines represent the average number of messages per second generated outside the burst.



(b) Traffic vs. reconfiguration rate.

Fig. 26. A comparison against event forwarding. The charts show the total message traffic generated, with a density of subscribers $\sigma_s = 0.8$.

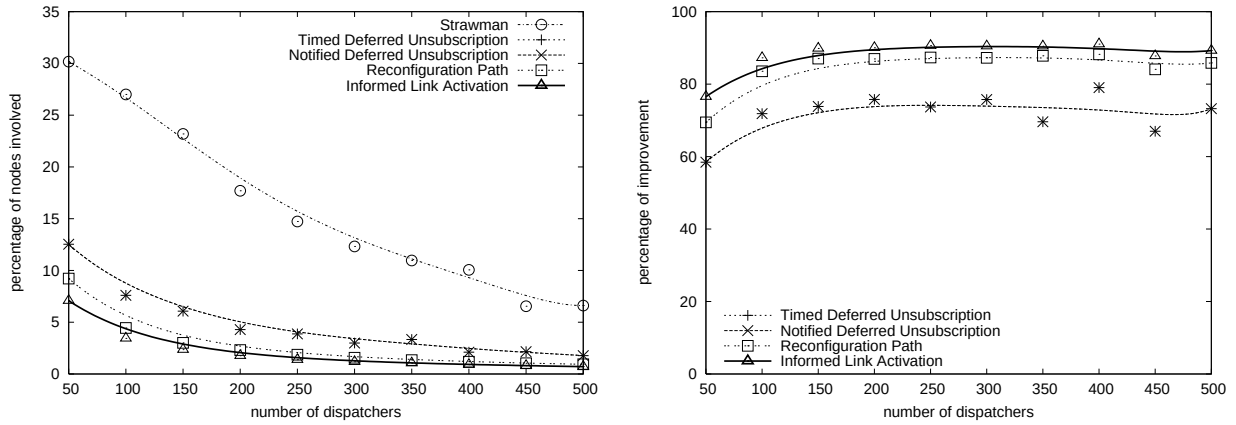
ACTIVATION protocol. The chart shows the traffic generated in a time interval between 2.5 and 8 seconds, with a burst of reconfigurations ($\rho = 30$ rec/s) that occurs in the interval $5s \leq t \leq 5.5s$. This yields 15 reconfigurations occurring in the aforementioned interval, which at $N = 100$ causes a significant disruption affecting most of the system. Moreover, to place event forwarding in a favorable scenario we assumed a high density of subscribers, i.e., $\sigma_s = 0.8$ and the standard publish frequency of $\varepsilon = 1$ pub/s. All the other parameters are unchanged from Table I. The horizontal lines represent the average number of messages per second generated by each protocol outside the reconfiguration burst. As shown in the chart, in a stable system the average traffic

is much higher for event forwarding, about twice as large as the one generated by subscription forwarding. However, during the reconfiguration burst in the plot, the overhead of the latter (albeit enhanced with the best of our protocols) becomes higher than for event forwarding.

The answer to our question above is therefore ultimately determined by the ratio between the reconfiguration rate ρ and the publish frequency ε. The higher the publish frequency, the higher the overhead induced by event forwarding. Consequently, if ε is high, ρ must also be very high to justify the use of event forwarding. The tradeoffs are exemplified by Figure 26(b), which shows the overall message traffic against ρ for $\varepsilon = 1$ pub/s, i.e., in the same conditions of Figure 26(a). If the value of ρ is reasonable (unlike those we used in the reconfiguration burst above or in Section V-C.2.d) then subscription forwarding is always preferable over event forwarding, with our protocols providing significant additional improvements.

5) Computational Overhead: Dispatchers Involved in a Reconfiguration: Thus far, we considered only the overall communication overhead induced by reconfiguration. Nevertheless, another way to look at the burden reconfiguration places on the system is to examine the computational overhead induced on the dispatchers. Our simulations do not capture this directly, and in any case the results would be too biased by the choice of the format of events and subscriptions. However, an indirect measure of the stress placed on the system is the (average) number of dispatchers involved in a single reconfiguration. In this section, we consider a dispatcher as *involved* in a reconfiguration if it performs processing related to the reconstruction of routes disrupted by it. Therefore, involved nodes include the end-points of the old and new link, as well as any other dispatcher processing subscriptions and unsubscriptions triggered by the reconfiguration, as well as REC messages. On the other hand, they do not include dispatchers that process *only* FLUSH messages, as the amount of processing they incur (rebroadcasting the message) is negligible if compared with the one for the aforementioned messages (manipulation of subscription tables, generation of new messages). Also, we do not include dispatchers that process misrouted events, as we want to characterize the overhead determined solely by the rebuilding of routes.

The value of this metric can be easily derived for each of the simulation traces presented thus far. In the following, for each chart we show on the left the absolute percentage of dispatchers involved in the reconfiguration, and on the right the percentage of improvement w.r.t. STRAWMAN. The results support the qualitative arguments put forth in Section IV, confirming that our protocols are able to significantly limit the portion of the system involved in a reconfiguration.



(a) Percentage of dispatchers involved.

(b) Percentage of improvement.

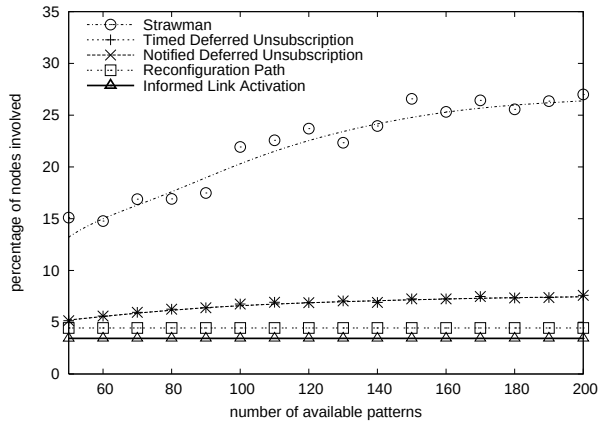
Fig. 27. Dispatchers involved in a reconfiguration vs. system scale.

For instance, the fraction of dispatchers involved is shown against system scale in Figure 27 using the same parameters of Figure 14. All of our protocols use less than half of the dispatchers used by STRAWMAN, with RECONFIGURATION PATH and INFORMED LINK ACTIVATION using 90% less than STRAWMAN. Note how the absolute percentage of dispatchers involved gets smaller as the scale is increased. This is a consequence of assuming a reconfiguration rate independent of the system scale ($\rho = 3$ rec/s in this case): as the scale increases, the disruption caused by each reconfiguration is amortized over a smaller fraction of the system. Moreover, additional simulations not shown here show that, similarly to what we discussed in Section V-C.2.a, an unbalanced initial configuration amplifies the differences among our approaches.

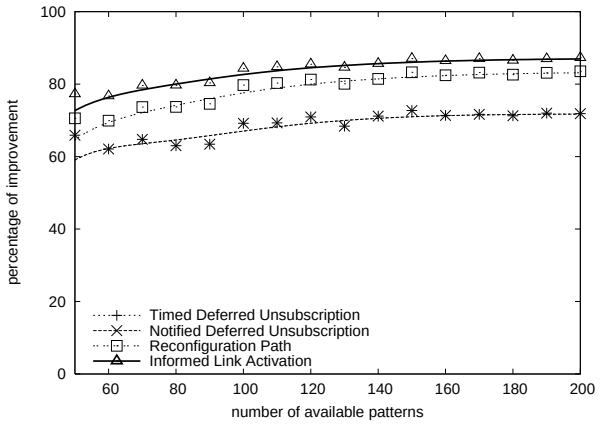
Another interesting perspective is provided by Figure 28 and 29, which plot the dispatchers involved against the density of subscribers and number of available patterns, respectively, with the same setting of Figure 16 and 17. These charts not only provide additional support for the ability of our protocols to limit reconfiguration, but also show how INFORMED LINK ACTIVATION, and even more RECONFIGURATION PATH, are largely independent of these two parameters.

VI. DISCUSSION AND LESSONS LEARNED

The previous section characterized the performance of our protocols along several dimensions, therefore providing a useful and immediate way to compare quantitatively the various solutions. Nevertheless, each protocol bears strengths and weaknesses, determined by the assumptions

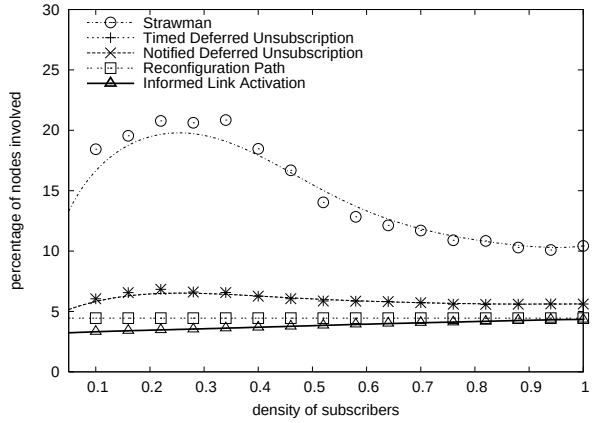


(a) Percentage of dispatchers involved.

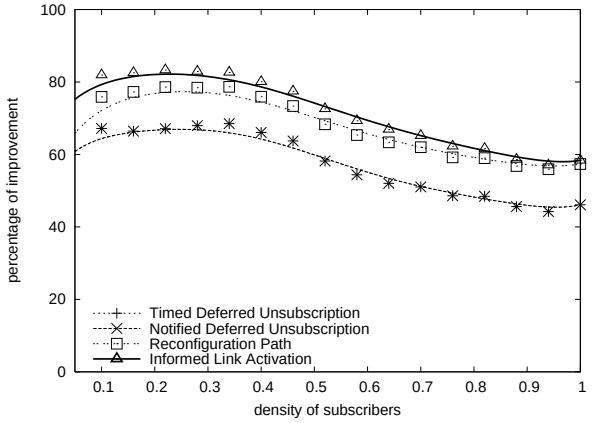


(b) Percentage of improvement.

Fig. 28. Dispatchers involved in a reconfiguration vs. number of available event patterns.



(a) Percentage of dispatchers involved.



(b) Percentage of improvement.

Fig. 29. Dispatchers involved in a reconfiguration vs. subscriber density.

it relies upon and by the very mechanics of its operations. As a consequence, it would be misleading to elect a single protocol as the best solution based uniquely on the simulation results of Section V. Instead, in this section we complement our quantitative results with qualitative considerations about the applicability and complexity of the solutions we presented. Together, these findings enable one to choose the most appropriate protocol for a given deployment scenario.

Figure 30 shows the main differences among the protocols we proposed in this paper. The first column refers to the ability of the protocol to tolerate multiple, concurrent reconfigurations,

		Arbitrary Changes?	*Knowledge of Topology*	*Additional Messages*	*Implementation Complexity*
STRAWMAN		Yes	A link (dis)appears	None	Low
DEFERRED UNSUBSCRIPTION	TIMED	Yes	A link (dis)appears	None	Low
	NOTIFIED	Yes	A link replaces another	FLUSH	Low/Medium
INFORMED LINK ACTIVATION		Yes	A link replaces another	ACTIVATE, FLUSH	Medium
RECONFIGURATION PATH		No	Reconfiguration path	REC, RECACK	High

Fig. 30. Comparing the applicability and complexity of reconfiguration protocols.

whose reconfiguration paths may or may not overlap. All protocols but RECONFIGURATION PATH have this capability. The second column characterizes the amount of knowledge each protocol assumes about the operation of the underlying tree maintenance module. The TIMED DEFERRED UNSUBSCRIPTION protocol, like STRAWMAN, only assumes that this sub-system is able to notify a dispatcher when one of its links disappears or when a new neighbor appears—a minimal assumption that can be satisfied straightforwardly. The NOTIFIED DEFERRED UNSUBSCRIPTION and INFORMED LINK ACTIVATION assume that the tree maintenance module is also able to determine and properly report whether the appearance of a new link is effectively a replacement of a given previously vanished link. This can be accomplished by specific overlay network protocols, such as those we developed recently [31], [25]. Finally, the RECONFIGURATION PATH protocol requires knowledge of the whole sequence of dispatchers belonging to the reconfiguration path, which again requires either dedicated protocols (e.g., straightforward adaptation of the aforementioned ones we developed) or more stringent assumptions on the deployment scenario (e.g., manual reconfiguration performed by a system administrator). Together, these two columns are a qualitative indicator of the applicability of the protocols to different scenarios.

The third column shows which control messages are necessary in each protocol, in addition to the messages normally used to deal with publish and (un)subscribe operations. As such it can be regarded as an indicator of the ease of implementation. The TIMED DEFERRED UNSUBSCRIPTION protocol affects only the order of the operations performed by STRAWMAN, and as such it does not introduce any new control messages, while all the other protocols introduce at least one. Finally, the fourth column summarizes the findings, by informally and

qualitatively classifying the protocols according to their overall implementation complexity. The TIMED DEFERRED UNSUBSCRIPTION protocol is by far the simplest, in that it makes simple (and yet effective) variations to STRAWMAN. At the other extreme, RECONFIGURATION PATH is by far the most complex, as evidenced by comparing its code and description to the other protocols in Section IV.

These qualitative considerations, together with the quantitative evaluation we presented in Section V, enable us to distill some conclusions. To begin with, TIMED DEFERRED UNSUBSCRIPTION is the least efficient of our protocols. Nevertheless, it still improves considerably over STRAWMAN and, like this protocol, it is applicable in virtually any environment. Moreover, its implementation is extremely simple. Therefore, TIMED DEFERRED UNSUBSCRIPTION is a viable solution when it is not necessary to fully optimize the reconfiguration traffic, but instead code footprint and applicability are more of a concern. The middle ground in terms of performance is occupied by the NOTIFIED DEFERRED UNSUBSCRIPTION alternative. This comes at the expense of a small increase in complexity and slightly more stringent assumptions about the overlay network, namely the ability to associate each new link with one that disappeared previously. The protocol generally provides a small and yet significant improvement over TIMED DEFERRED UNSUBSCRIPTION. The best performance among the four alternatives to STRAWMAN is provided by RECONFIGURATION PATH and INFORMED LINK ACTIVATION. The performance of the former, however, comes at a significant cost, which makes its value more theoretical than practical. The protocol is, in fact, characterized by a very high implementation complexity, is unable to withstand multiple overlapping reconfigurations, and poses significant requirements on the underlying tree maintenance protocols. The latter protocol, on the other hand, is able to achieve a comparable performance in arbitrary reconfiguration scenarios, and with lower requirements on the tree maintenance module. These requirements are the same as those posed by the NOTIFIED DEFERRED UNSUBSCRIPTION protocol; as a result INFORMED LINK ACTIVATION strikes the best tradeoff between performance and applicability.

VII. RELATED WORK

In this section we discuss related work in the field of content-based publish-subscribe and other paradigms for multi-point communication and coordination.

A. Distributed Content-based Publish-subscribe

Initial experiences with publish-subscribe focused on local area networks and relied on a centralized dispatcher, while recent years have seen the development of a number of content-based publish-subscribe systems based on a distributed dispatcher. Among the most widely known are Siena [8], Gryphon [4], Hermes [38], [39], Xnet [11], REBECA [24], Jedi [17], Joram [3], NaradaBrokering [36], Le Subscribe [22], READY [26], Elvin [42], and TIBCO Rendezvous [46].

Most of these systems do not provide any explicit mechanism to reconfigure the dispatching infrastructure in reaction to changes in the underlying network. Some, including Jedi [17], the extended version of Siena presented in [7], Elvin [45], REBECA [33], [23], and the work described in [40] support a different reconfiguration scenario where clients are enabled to roam by detaching from one dispatcher and attaching to another. However, none of these works address the reconfiguration of the dispatching network itself.

One type of exception is provided by Siena [8] and the system described in [51], which briefly mention the use of the STRAWMAN protocol to allow sub-trees of dispatchers to be merged or trees to be split. Neither of these papers provide details about the design of this facility, nor assess its effectiveness through simulation.

A different solution to the problem of rearranging the dispatching network is provided by Hermes [38], [39], which offers a slightly limited form of content-based routing, termed “type and attribute based” routing [21]. Hermes borrows techniques from the area of peer-to-peer systems to organize its dispatchers as a distributed hash table, where keys represent event types and key-based routing techniques are used to build the dispatching tree associated to each event type. This way, the issues stemming from the dynamic addition, removal, and failure of dispatchers are automatically handled by the peer-to-peer overlay underlying Hermes. However, the overhead generated by this approach has not been analyzed to date.

In our research we adopted a reactive approach, based on the idea of rearranging the routing table upon a change in the overlay network. Instead, some recent approaches are based on a proactive and periodic (re)propagation of subscriptions. In [32], subscriptions are associated to a lease and must be refreshed directly by the clients; subscriptions whose lease expires are automatically discarded by dispatchers. Instead, in [9] a dedicated protocol is run periodically

to remove stale subscriptions generated by reconfigurations. Both protocols deal with dynamic topologies at the cost of requiring a continuous re-propagation of subscriptions. In these proposals, however, the frequency of refresh is a critical parameter difficult to tune. If it is too low the system is not responsive to changes and the stale routes may contribute to a large number of misrouted events. If it is too high, the overhead becomes unbearable. A reactive approach such as the one we adopted is not affected by these problems, as it shows only a limited dependence on the reconfiguration rate.

JEcho [12] takes a different approach in the context of mobile ad hoc networks (MANETs) by assuming the availability of a multi-hop unicast protocol. The latter is used to maintain a tree-shaped overlay for routing publish-subscribe messages, and essentially mask the topology changes induced by mobility. This approach simplifies the task of the publish-subscribe system, which can be designed as if it would operate on a fixed network, but may easily result in an overlay topology that rapidly diverges from the physical one. Therefore, JEcho dispatchers periodically run a link state protocol to build a global view of the physical network and rearrange the overlay accordingly. The overhead of this mechanism is analyzed in the paper. Unfortunately, the authors fail to provide details about how to restore correct routing tables when a topological reconfiguration is forced to realign the overlay with the physical network. Therefore, the solution described in [12] is essentially concerned only with the optimization of the tree overlay, and is complementary to the work we described here.

Some approaches have also tried to move away from the presence of a tree-shaped overlay. For instance, the work in [16] exploits the redundancy of a graph overlay through a mixture of deterministic (i.e., based on subscriptions) and probabilistic routing, where the latter component intrinsically provides enhanced resilience to reconfiguration. Others, and especially those conceived in the field of MANETs [53], [30], [50], [2], do not maintain an overlay, instead they rely directly on broadcast communication. In general, all these works are quite far from the one described here, and cannot be immediately compared to it.

Finally, in addition to the systems mentioned thus far, that consider a reconfiguration problem similar to ours, other systems focus on fault-tolerance and reliability. In particular, Xnet [11] provides several mechanisms, including the use of redundant routes, to reduce the impact of dispatcher crashes. Analogously, Joram 4.2 [3] and the extension to Gryphon described in [52] allow a set of dispatchers to operate as a single redundant cluster to deal with link failures and

dispatcher crashes. A similar approach is also discussed in [43] in the context of MANETs. Multiple dispatchers operate as replicated servers, i.e., by sharing the subscriptions issued by all clients on the system. Published messages are then propagated to all servers using an approach analogous to the event forwarding strategy we discussed. In [28] several alternatives to adapt publish-subscribe systems to mobile middleware are briefly discussed. None of them is detailed, not to say evaluated. Replication is also discussed as a way to increase the availability and reliability of a publish-subscribe service in presence of mobility, e.g., to overcome server failures or network partitions. However, all these approaches provide a limited form of reconfigurability with respect to the one offered by the protocols studied here.

B. Network-level Multicast and Group Communication

An alternative perspective on publish-subscribe is to view it as a form of multi-point communication. Therefore, it is natural to investigates the relations of our work with other forms of multi-point communication, e.g., multicast network protocols and group communication middleware.

The purpose of network-level multicast (e.g., as implemented over IP) is to provide efficient datagram communication services for applications that need to send the same data to a group of recipients. Typical examples are audio and video streaming. This goal results in routing strategies that largely differ from the ones adopted in publish-subscribe systems. In particular, applications based on multicast exploit a finite number of statically known groups, while in content-based publish-subscribe systems the “groups” (i.e., the event patterns) are potentially infinite and not statically known. Moreover, IP multicast groups are disjoint and each packet is explicitly addressed to a single group. Instead, in the systems we considered, addressing is based on event content, therefore a event can match (and be routed based on) multiple subscriptions. Finally, publish-subscribe usually assumes the number of sources to be comparable to (if not much greater than) the number of recipients, while multicast protocols are often devised to satisfy a small set of sources communicating with a large set of recipients. As a consequence of these differences, it is not practical to generalize IP multicast routing protocols to route events in a content-based publish-subscribe system. For similar reasons, it is hard to implement a content-based publish-subscribe system on top of an existing IP multicast protocol. This issue is discussed in detail in [35], where several alternatives are compared.

Instead, the term “group communication” identifies a body of research whose goal is to

provide mechanisms for reliable communication among a group of possibly remote processes, guaranteeing some properties about event ordering and atomicity [13]. Under this umbrella fall systems providing reliable multicast [29], [34] as well as systems providing semantically richer functionality to coordinate a set of distributed components [5], [48]. As in network-level multicast, in these systems it is the sender that determines the set of recipients, in contrast with content-based publish-subscribe systems where it is the receiver that specifies the event classes of interest. Moreover, while group communication systems provide reliability, ordering, and atomicity, content-based publish-subscribe systems emphasize performance and scalability at the cost of providing less guarantees. This different focus has a strong impact on the underlying protocols and mechanisms adopted, therefore the solutions developed for group communication are not directly reusable in our context.

VIII. Conclusions and Future Work

Content-based publish-subscribe systems have become increasingly popular in recent years thanks to the high level of flexibility they bring in the development of distributed applications. Although much effort has been devoted to the design of scalable solutions supporting publish-subscribe middleware in large-scale scenarios, existing systems still lack efficient ways to address changes in the topology of their distributed dispatching infrastructure.

Supporting this functionality requires addressing different problems. In this paper, we focused on the issue that is peculiar to content-based systems: how to efficiently reconcile the information used to route events to subscribers in the presence of topological reconfigurations. We presented our overall approach to the reconfiguration problem and described four protocols that extend the common subscription forwarding strategy with the ability to tolerate topology changes in a number of application scenarios. The protocols were thoroughly compared with extensive simulation experiments. These results allowed us to assess the advantages and drawbacks of each solution, providing valuable information to middleware designers. Depending on the specific scenarios, our protocols manage to reduce the overhead of the reconfiguration process by up to 80% without hampering the ability to deliver event notifications to interested parties.

The protocols we presented are currently being integrated in our content-based publish-subscribe middleware REDS (REconfigurable Dispatching System) [19], available as open source at `zeus.elet.polimi.it/reds`. This will enable a more direct assessment of their perfor-

mance in real-world applications. Moreover, we are currently working on evaluating our protocols when used in conjunction with our solutions for repairing the overlay [25], [31] and recovering lost events [14], [15]. The combination of these protocols with those specific to routing described in this paper is expected to evidence additional tradeoffs and possibly further opportunities for optimization.

Acknowledgments: The work described in this paper was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, by the National Research Council (CNR) under the IS-MANET project, and by the European Community under the IST-004536 RUNES project.

REFERENCES

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communication Mag.*, 40(8):102–114, 2002.

[2] R. Baldoni, R. Beraldi, G. Cugola, M. Migliavacca, and L. Querzoni. Structure-less content-based routing in mobile ad hoc networks. In *Proc. of the IEEE Int. Conf. on Pervasive Services*, Santorini (Greece), July 2005. IEEE Computer Society Press.

[3] R. Balter. Joram: The open source enterprise service bus. Technical report, ScalAgent Distributed Technologies, March 2004. `www.scalagent.com/pages/en/datasheet/040322-joram-whitepaper-en.pdf`.

[4] 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 19^{th} Int. Conf. on Distributed Computing Systems*, Austin, TX, USA, 1999. IEEE Computer Society Press.

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

[6] Fengyun Cao and Jaswinder Pal Singh. Efficient event routing in content-based publish/subscribe service network. In *INFOCOM*, 2004.

[7] M. Caporuscio, A. Carzaniga, and A.L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. on Software Engineering*, 29(12):1059–1071, December 2003.

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

[9] A. Carzaniga, M.J. Rutherford, and A.L. Wolf. A Routing Scheme for Content-Based Networking. In *Proc. of the 23rd Confe. of the IEEE Communications Society (INFOCOM'04)*, Hong Kong, China, March 2004. IEEE Computer Society Press.

[10] R. Chand and P.A. Felber. A scalable protocol for content-based routing in overlay networks. In *Proc. of the 2nd IEEE Int. Symp. on Network Computing and Applications*, page 123, Cambridge, MA, April 2003. IEEE Computer Society Press.

[11] R. Chand and P.A. Felber. XNet: a reliable content based publish subscribe system. In *Proc. of the 23rd Symp. on Reliable Distributed Systems*, Florianópolis, Brazil, Oct 2004. IEEE Computer Society Press.

[12] Y. Chen and K. Schwan. Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In G. Alonso, editor, *Proc. of the 6th ACM/IFIP/USENIX Int. Middleware Conf.*, LNCS 3790, pages 354–374, Grenoble, France, November 2005. Springer.

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

[14] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Introducing Reliability in Content-Based Publish-Subscribe through Epidemic Algorithms. In *Proc. of the 2^{nd} Int. Workshop on Distributed Event-Based Systems (DEBS'03)*, San Diego, CA, USA, June 2003. ACM Press.

[15] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proc. of the 24^{th} Int. Conf. on Distributed Computing Systems (ICDCS'04)*, pages 552–561, Tokyo, Japan, march 2004. IEEE Computer Society Press.

[16] P. Costa and G. P. Picco. Semi-probabilistic Content-based Publish-subscribe. In *Proc. of the 25^{th} Int. Conf. on Distributed Computing Systems (ICDCS05)*, pages 575–585, Columbus (OH, USA), June 2005. IEEE Computer Society Press.

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

[18] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the Reconfiguration Overhead in Content-Based Publish-Subscribe. In *Proc. of the 19^{th} ACM Symp. on Applied Computing (SAC'04)*, pages 1134–1140, Nicosia, Cyprus, 2004. ACM Press.

[19] G. Cugola and G. P. Picco. REDS: A Reconfigurable Dispatching System. Technical report, Politecnico di Milano, March 2005. Submitted for publication. `www.elet.polimi.it/upload/picco`.

[20] G. Cugola, G. P. Picco, and A. L. Murphy. Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Systems. In A. Coen-Porisini and A. van Der Hoek, editors, *Proceedings of the 3^{rd} International Workshop on Software Engineering and Middleware (SEM'02), co-located with the 24^{th} International Conference on Software Engineering (ICSE'03)*, volume 2596 of *Lecture Notes on Computer Science*, pages 187–202, Orlando (FL, USA), May 2002. Springer.

[21] P.T. Eugster, R. Guerraoui, and C.H. Damm. On objects and events. In *Proc. of the OOPSLA'01 Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 254–269, Tampa Bay (FL, USA), October 2001.

[22] F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *ACM SIGMOD Record*, 30(2):115–126, 2001.

[23] L. Fiege, F.C. Gartner, O. Kasten, and A. Zeidler. Supporting Mobility in Content-Based Publish/Subscribe Middleware. In *Proc. of the 4th ACM/IFIP/USENIX Int. Middleware Conf.*, Rio de Janeiro, Brazil, June 2003. ACM Press.

[24] L. Fiege, G. Mühl, and F.C. Gärtner. Modular event-based systems. *Knowledge Engineering Review*, 17(4):359–388, 2002.

[25] Davide Frey and Amy L. Murphy. Failure-tolerant overlay trees for large-scale dynamic networks. Technical report, 2008.

[26] R.E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In P. Dasgupta, editor, *Proc. of the ICDCS Workshop on Electronic Commerce and Web-Based Applications*, Austin, TX, USA, May 1999. IEEE Computer Society Press.

[27] Y. Huang and H. Garcia-Molina. Publish/Subscribe Tree Construction in Wireless Ad-Hoc Networks. In *Proc. of the 4^{th} Int. Conf. on Mobile Data Management (MDM '03)*, pages 122–140, Melbourne, Australia, 2003. Springer.

[28] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile enviroment. In *MobiDe01: Proc. of the 2^{nd} ACM Int. Workshop on Data engineering for Wireless and Mobile access*, pages 27–34, Santa Barbara, CA, 2001. ACM Press.

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

[30] René Meier and Vinny Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Network. In *Proc. of the* 22^{nd} *Int. Conf. on Distributed Computing Systems*, pages 639–644, Vienna, Austria, 2002. IEEE Computer Society.

[31] L. Mottola, G. Cugola, and G. P. Picco. Tree-based overlays for publish-subscribe in mobile ad hoc networks. Technical report, Politecnico di Milano, 2005. Submitted for publication. `www.elet.polimi.it/upload/picco`.

[32] G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, and A. Ulbrich. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In J.C. Cunha and P.D. Medeiros, editors, *Proc. of the 11th European Conference on Parallel Computing (EuroPar 2005)*, LNCS 3684, Lisboa, Portugal, August 2005. Springer.

[33] Gero Mühl, Andreas Ulbrich, Klaus Herrmann, and Torben Weis. Disseminating information to mobile clients using publish/subscribe. *IEEE Internet Computing*, 8(3):46–53, May 2004.

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

[35] L. Opyrchal et al. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proc. of Middleware 2000*, New York, USA, 2000. ACM Press.

[36] S. Pallickara and G. Fox. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proc. of the 4th ACM/IFIP/USENIX Int. Middleware Conf.*, pages 41–61, Rio de Janeiro, Brazil, June 2003. ACM Press.

[37] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient Content-Based Event Dispatching in Presence of Topological Reconfiguration. In *Proc. of the* 23^{rd} *Int. Conf. on Distributed Computing Systems (ICDCS'03)*, pages 234–243, Providence, Rhode Island, USA, May 2003. IEEE Computer Society Press.

[38] P.R. Pietzuch and J.M. Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the* 1^{st} *Int. Workshop on Distributed Event-Based Systems (DEBS)*, Vienna, Austria, July 2002. IEEE Computer Society Press.

[39] P.R. Pietzuch and J.M. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *Proc. of the* 2^{nd} *Int. Workshop on Distributed Event-Based Systems (DEBS)*, San Diego, CA, USA, June 2003. ACM Press.

[40] I. Podnar and I. Lovrek. Supporting mobility with persistent notifications in publish-subscribe systems. In *Proc. of the* 3^{rd} *Int. Workshop on Distributed Event-Based Systems (DEBS)*, Edinburgh, Scotland, UK, May 2004. ACM Press.

[41] Leon Poutievski, Kenneth L. Calvert, and James Griffioen. Speccast. In *INFOCOM*, 2004.

[42] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proc. of AUUG2K*, Canberra, Australia, June 2000.

[43] Katrine Stemland Skjelsvik, Vera Goebel, and Thomas Plagemann. Distributed event notification for mobile ad hoc networks. *IEEE DSOnline*, 5(8), 2004.

[44] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-based content routing using xml. *SIGOPS Oper. Syst. Rev.*, 35(5):160–173, 2001.

[45] 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. IEEE Computer Society Press.

[46] TIBCO Inc. TIBCO Rendezvous. `www.tibco.com`.

[47] P. Triantafillou and A. Economides. Subscription summarization: A new paradigm for efficient publish/subscribe systems.

In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, march 2004. IEEE Computer Society Press.

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

[49] A. Varga. OMNeT++ Web page, 2003. `www.omnetpp.org`.

[50] E. Yoneki and J. Bacon. An adaptive approach to content-based subscription in mobile ad hoc networks. In *Proc. of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops*, Orlando, FL, March 2004. IEEE Computer Society Press.

[51] 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*, Stanford, CA, USA, 1999. IEEE Computer Society Press.

[52] Y. Zhao, D. Sturman, and S. Bhola. Subscription propagation in highly-available publish/subscribe middleware. In *Proc. of the 5th ACM/IFIP/USENIX Int. Conf. on Middleware*, pages 274–293, Toronto, Canada, 2004. Springer.

[53] H. Zhou and S. Singh. Content-based multicast for mobile ad hoc networks. In *Proc. of the 1st Annual Workshop on Mobile Ad Hoc Networking and Computing (Mobihoc 2000)*, Boston, MA, Aug 2000. ACM Press.