Publish Subscribe on Large-Scale Dynamic Topologies: Routing and Overlay Management

Davide Frey

▶ **To cite this version:**

Davide Frey. Publish Subscribe on Large-Scale Dynamic Topologies: Routing and Overlay Management. Distributed, Parallel, and Cluster Computing [cs.DC]. Politecnico di Milano, 2006. English. NNT : . tel-00739652

HAL Id: tel-00739652
https://theses.hal.science/tel-00739652v1

Submitted on 8 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Publish Subscribe on Large-Scale Dynamic Topologies: Routing and Overlay Management

Doctoral Dissertation of:
Davide Frey

Advisor:
Prof. Gian Pietro Picco

Coadvisor:
Prof. Amy L. Murphy

Tutor:
Prof. Elisabetta Di Nitto

Supervisor of the Doctoral Program:
Prof. Stefano Crespi Reghizzi

2006 - XVIII

To my father

Acknowledgements

Many people played an important role in directing me towards completing this thesis: some with their guidance, some with their advice, some with their care, and some simply by bearing with me in the most difficult moments. I would like to thank them all.

First, I wish to thank my advisors, Gian Pietro Picco and Amy Murphy, for their guidance, their remarks, and their collaboration in the work I carried out during my doctoral studies. Together with Gianpaolo Cugola, they introduced me to research, contributing to my professional growth, but also playing an important role as friends. I wish to thank Antonio Carzaniga, for the comments he made in his review of this thesis. I want to thank, Federico Malucelli, for his advice, and enthusiasm in guiding me through my minor research; and also Carlo Ghezzi and Elisabetta Di Nitto for their constant support and advice.

So much for big names; now the bigger ones: I wish to thank Paolo, for his collaboration, advice, and mostly for his friendship, and for not complaining about our untidy shared desk. My gratitude then goes to the many other colleagues with whom I have shared interests, time, space, food, and laughter: Sam, Luca, Davide, Matteo M., Matteo P., Angelo, Roberto, Silvana, Vincenzo, Mattia, Sabrina, Paola, Alessandro, Carlo Alberto, Marco. Here I am also grateful to my doctoral studies, for giving me the opportunity to meet and interact with these wonderful people.

Lastly, but most importantly I would like to thank people outside my professional environment. First I wish to express my gratitude to Clara, for her constant support, for sharing with me the most beautiful moments and for encouraging me in the hardest ones. I then wish to thank my family: my mother, and my sister for their constant support, understanding, and patience. Finally, I am grateful to the many friends who have always been there for me even the busiest and most difficult time of my studies: Liliana, Lucia, Franco to name only a few.

Thank you all!!!

Abstract

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

This thesis addresses this problem and presents a complete approach to the reconfiguration of content-based publish-subscribe systems. In Part I, it proposes a layered architecture for reconfigurable publish-subscribe middleware consisting of an overlay, a routing, and an event-recovery layer. This architecture allows the same routing components to operate in different types of dynamic network environments, by exploiting different underlying overlays.

Part II addresses the routing layer with new protocols to manage the reconfiguration of the routing information enabling the correct delivery of events to subscribers. When the overlay changes as a result of nodes joining or leaving the network or as a result of mobility, this information is updated so that routing can adapt to the new environment. Our protocols manage to achieve this with as little overhead as possible.

Part III addresses the overlay layer and proposes two novel approaches for building and maintaining a connected topology in highly dynamic network scenarios. The protocols we present achieve this goal, while managing node degree and keeping reconfigurations localized when possible. These properties allow our overlay managers to be applied not only in the context of publish-subscribe middleware but also as enabling technologies for other communication paradigms like application-level multicast.

Finally, the thesis integrates the overlay and routing layers into a single framework and evaluates their combined performance both in wired and in wireless scenarios. Results show that the optimizations provided by our routing reconfiguration protocols allow the middleware to achieve very good performance in such networks. Moreover, they highlight that our overlay layer is able to optimize this performance even further, significantly reducing the network traffic generated by the routing layer.

The protocols presented in this thesis are implemented in the REDS middleware framework developed at Politecnico di Milano. Their use enables REDS to operate efficiently in dynamic network scenarios ranging from large-scale peer-to-peer to mobile ad hoc networks.

Riassunto

I sistemi publish-subscribe con approccio content-based stanno affermando il proprio paradigma di comunicazione come in grado di soddisfare i requisiti di applicazioni distribuite in scenari altamente dinamici come quelli resi possibili da reti mobili e peer-to-peer. Ci nonostante, i sistemi esistenti basati sul paradigma publish-subscribe sono molto spesso ancora incapaci di gestire in modo efficiente i cambiamenti nella topologia del sistema di distribuzione dei messaggi. Questo limita la loro applicazione in scenari di rete dinamici.

Questa tesi di dottorato affronta queste problematiche e propone un approccio completo per la realizzazione di middleware publish-subscribe riconfigurabile. Dapprima viene proposta un'architettura a livelli comprendente uno strato di overlay, uno strato di routing ed uno strato di event recovery. Quest'architettura consente al middleware di impiegare gli stessi meccanismi di routing in diversi scenari applicativi, semplicemente cambiandone lo strato di overlay.

La tesi quindi affronta le problematiche relative allo strato di routing con protocolli per gestire la riconfigurazione delle informazioni di routing che consentono la consegna dei messaggi ai componenti applicativi interessati a riceverli. Quando la struttura dell'overlay cambia in seguito a disconnessioni o alla mobilità dei nodi della rete, le informazioni di routing vengono aggiornate ed adattatte alla nuova topologia. I protocolli proposti in questa tesi consentono di effettuare queste operazioni con un costo minimo.

Il passo successivo affronta lo strato di overlay proponendo due meccanismi per la costruzione ed il mantenimento di una topologia ad albero in scenari fortemente dinamici. I protocolli presentati raggiungono quest'obiettivo, limitando il numero di vicini di ciascun nodo e localizzando le riconfigurazioni. Queste proprietà consentono l'applicazione dei nostri protocolli non solo nel contesto del middleware publish-subscribe ma anche come tecnologie base per altri paradigmi di comunicazione, o per sistemi di multicast a livello applicativo.

La tesi si conclude con un'integrazione degli strati di overlay e routing in un unico framework. Ci consente di valutarne le prestazioni in scenari peer-to-peer ed in reti ad hoc. I risultati mostrano che le ottimizzazioni messe in atto dai nostri protocolli consentono al middleware di ottenere ottime prestazioni. Inoltre mostrano come i protocolli per la gestione dell'overlay siano in grado di ottimizzare le prestazioni del middleware, riducendo ulteriormente il traffico generato dallo strato di routing.

I protocolli presentati in questa tesi sono implementati nel middleware REDS sviluppato presso il Politecnico di Milano. Il loro impiego consente a REDS di operare in maniera efficiente in sia scenari peer-to-peer che in reti ad hoc.

CONTENTS

1 **Introduction** 1

I Problem Definition 5

2 **Background** 7
- 2.1 The Publish-Subscribe Paradigm . . . 8
- 2.2 Publish-Subscribe . . . 8
- 2.3 Publish-Subscribe Extensions . . . 10
- 2.4 Implementation of Publish-Subscribe . . . 12
  - 2.4.1 Architecture of the Event Dispatcher . . . 12
  - 2.4.2 Matching Events against Subscriptions . . . 16
- 2.5 Existing Publish Subscribe Middleware . . . 16

3 **Motivation and Approach** 19
- 3.1 Targeting Dynamic Scenarios . . . 20
- 3.2 Existing Solutions . . . 21
- 3.3 Reconfigurable Tree-Based Publish-Subscribe . . . 22
  - 3.3.1 Key Design Choices . . . 22
  - 3.3.2 A Reconfigurable Middleware Architecture . . . 23
  - 3.3.3 Addressing Different Failure Models . . . 26
- 3.4 Concluding Remarks . . . 28

II Routing Layer 29

4 **Routing Requirements and Goals** 31

5 **Baseline and Observations** 33
- 5.1 Baseline: Strawman Protocol . . . 33
- 5.2 Understanding Propagation and Reconfiguration . . . 34

6 Optimized Protocols **41**
6.1 Deferred Unsubscription 41
6.1.1 Timed Deferred Unsubscription 42
6.1.2 Reducing the Dependence on Timers: Notified Deferred Unsubscription 44
6.2 Informed Link Activation 45
6.3 Reconfiguration Path 48
6.3.1 Basic Operation 50
6.3.2 Dealing with Concurrent (Un)Subscriptions 51

7 Evaluation of Routing Reconfiguration **53**
7.1 Simulation Setting 53
7.2 Event Delivery 55
7.3 Overhead 56
7.3.1 Modeling the Cost of Publish-Subscribe and Control Messages 57
7.3.2 Evaluating the Cost of Reconfiguring Subscription Tables . 59
7.3.3 Evaluating the Impact of Misrouted Events 66
7.3.4 A Note About Event Forwarding 69
7.3.5 Computational Overhead: Dispatchers Involved in a Reconfiguration 70

8 Discussion and Related Work **73**
8.1 Applicability Versus Performance 73
8.2 Related Work 76
8.2.1 Distributed publish-subscribe systems 76
8.2.2 Publish-subscribe on MANETs 77
8.2.3 Other research areas 78
8.3 Concluding Remarks 79

III Overlay Layer **81**

9 Overlay Requirements and Goals **83**

10 Large-Scale Tree Maintenance **87**
10.1 High Level Maintenance Protocol 87
10.2 Determining Parent Viability 89
10.2.1 Hard Requirement: single acyclic tree 89
10.2.2 Soft Requirements on Node Degree 92
10.3 Identifying Candidate Parents 92
10.3.1 Repair Strategies 93
10.3.2 Combining Recovery Techniques 95
10.3.3 Declaring New Roots and Merging Trees 95
10.4 Simulation Study 96
10.4.1 Simulation Setting 96
10.4.2 Results 97
10.5 Concluding Remarks 103

11 A DHT-based Overlay 105
11.1 Distributed Hash Tables . . . 105
11.2 Protocol Description . . . 106
11.2.1 Reference Tree Structure . . . 107
11.2.2 Determining a Node's Neighborhood . . . 108
11.2.3 Maintaining Overlay Properties . . . 110
11.2.4 Example . . . 111
11.3 Evaluation . . . 112
11.3.1 Simulation Setting . . . 112
11.3.2 Results . . . 113
11.4 Concluding Remarks . . . 115

12 Discussion and Related Work 117
12.1 Two opposite approaches . . . 117
12.2 Overlay in Publish Subscribe . . . 119
12.3 Related Approaches . . . 120
12.3.1 Application-Level Multicast . . . 120
12.3.2 Mobile Ad Hoc Multicast . . . 121
12.3.3 Overlays for Publish-Subscribe Systems . . . 121
12.3.4 Distributed Tree and Spanning Tree Construction . . . 122
12.4 Concluding Remarks . . . 123

IV Putting It All Together 125

13 Integration Requirements and Goals 127

14 Towards an Integrated Evaluation 129
14.1 Integrating the Overlay Layer . . . 129
14.1.1 Choice of the Wired Overlay . . . 130
14.1.2 Choice of the Wireless Overlay . . . 130
14.2 Integrating the Routing Layer . . . 131
14.2.1 Choice of the Routing Reconfiguration Protocols . . . 132
14.2.2 Integrating STRAWMAN and TIMED DEFERRED UNSUBSCRIPTION . . . 132
14.2.3 Integrating INFORMED LINK ACTIVATION . . . 133
14.3 Integrated Simulation Framework . . . 136

15 Evaluation in P2P Scenarios 139
15.1 Simulation Setting . . . 139
15.2 Simulation Results . . . 140
15.2.1 Routing Reconfiguration . . . 141
15.2.2 Routing Reconfiguration Versus Overlay Maintenance . . . 144
15.2.3 Impact of Optimized Protocols . . . 145
15.3 Concluding Remarks . . . 148

16 Evaluation in MANETs **149**
16.1 Simulation Setting 149
16.2 Simulation Results 150
16.2.1 Routing Reconfiguration 150
16.2.2 Routing Reconfiguration versus Overlay Maintenance . . . 152
16.2.3 Comparison with Flooding 153
16.3 Concluding Remarks 156

Conclusions **157**

CHAPTER
ONE

Introduction

Recent trends in computing have shown an increase in distributed applications operating in dynamic environments such as wide area networks involving large numbers of nodes and multiple service providers. The physical infrastructure enabling these applications is however inherently different from what it was in the original design of the Internet. Modern personal computers or even palmtop devices are several times as powerful as the devices that made up the Internet in its early days, and broadband connections are rapidly becoming a standard not only for enterprises but also in most people's homes.

Problems arising from the unreliability of communication are made worse by the fact that hosts themselves join and leave the network at unpredictable moments. The model based on a network of servers whose disconnection is a rare event almost always due to failures therefore does not hold anymore. Nevertheless, a growing set of applications ranging from cycle scavenging to file sharing challenge these scenarios and seek to form communities out of these dynamic groups. Middleware researchers and designers have joined this challenge, and seek to support application developers with abstractions facilitating the development of distributed computing services.

Publish Subscribe

The publish-subscribe communication paradigm with the strong decoupling it introduces between the components of distributed applications appears as a very promosing tool to address these new scenarios. In publish-subscribe, 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 clients to specify these classes using linguistic facilities that 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 imple-

mentations, 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 [18], Jedi [35], Gryphon [11], and many others.

Despite the extreme flexibility of the paradigm, currently available publish-subscribe middleware offers only limited support for emerging dynamic scenarios. Some middleware systems allow client components to move from one broker to another in different logical or physical locations, while others rely on redundant topologies or on periodically refreshing routing information. However, the underlying model is very often still based on a stable network of servers whose disconnection is a rare event. Even systems that support dynamic environments do so only with basic protocols, characterized by a significant reconfiguration cost.

The next challenge for publish-subscribe middleware is therefore to tolerate dynamic scenarios by reconfiguring the operation of the distributed dispatching infrastructure in an efficient way. The motivations are numerous. For example, peer-to-peer networks are defining very fluid application-level networks for information sharing and dissemination, while mobility is increasingly becoming part of mainstream computing. The very characteristics of the publish-subscribe *model*, most prominently the sharp decoupling it introduces between communication parties, make it amenable to these and other highly dynamic environments. However, this is true in practice *only* if the publish-subscribe *system* is itself capable of dealing with reconfiguration. Unfortunately, the majority of the systems available in the literature do not provide such support. Filling this gap is precisely the goal of the work described in this thesis.

Approach

In the work presented in this thesis, we focus on distributed content-based publish-subscribe middleware based on a tree dispatching topology. The reason for this choice is twofold. First, trees represent the most efficient solution for data dissemination. Second, a large number of existing systems are based on tree topologies and thus the choice of trees makes our work directly applicable to existing publish-subscribe middleware.

With our approach, we also confront the one taken by some parallel efforts for the design of middleware for dynamic environment. The lack of redundant paths, an essential characteristic of trees, has lead these efforts to focus primarily on mesh topologies, constructing data distribution trees as subgraphs of the more connected mesh overlays. In this thesis we show that sacrificing efficient communication for redundant paths is not necessary even in the presence of frequent failures. Specifically, we show that systems based on tree overlays can be efficiently reconfigured in the face of frequent changes in the set of devices constituting the network as well as in the interconnections between them.

To achieve this, we address the reconfiguration of the topology of the distributed event dispatcher at the basis of publish-subscribe middleware. Dealing with topological reconfiguration is a multi-faceted problem, involving restoring the connectivity of the overlay network containing the dispatching servers, recovering the events lost during reconfiguration, and restoring the consistency of the

routing information steering events towards subscribers. In this thesis we address these issues with a layered middleware architecture consisting of an overlay, a routing, ad an event-recovery layer.

The overlay layer is responsible for building and maintaining a connected overlay topology in highly dynamic network scenarios. The routing layer operates above the overlay and manages the delivery of information to the correct recipients. Finally, the event-recovery layer addresses the recovery of events that may be lost during reconfiguration.

Contribution of the Thesis

The contribution of this thesis covers two of the layers of this middleware architecture: overlay and routing. At the overlay layer, we designed two protocols for managing a tree topology in large-scale wired networks. Both provide very good performance not only in the context of publish-subscribe middleware but also as enabling technologies for other communication paradigms like application-level multicast.

At the routing layer, we developed protocols to reconfigure routing when the underlying overlay topology changes. In such cases the routing layer must reconfigure its operation to adapt to the new environment with as little overhead as possible. In this thesis we propose two new solutions that significantly improve the performance of the basic protocols presented in current literature.

In addition, this thesis provides a contribution in terms of the evaluation of the reconfiguration protocols at the overlay and routing layers. The protocols proposed in the thesis are validated by means of extensive simulation studies that compare them against other solutions described in the literature. The detailed analysis of the solutions in the literature constitutes by itself a contribution.

The protocols for the routing and overlay layers are first evaluated individually and then in an integrated simulation framework. This integrated evaluation highlights the interactions between the protocols we designed and shows how each can affect the performance of the others. As often happens in engineering complex systems, results show that protocols providing smaller improvement but wider applicability are often preferable to protocols performing better in more constrained situations.

Finally, our analysis shows that efficient publish-subscribe middleware for dynamic network environments is a reality. The solutions we present are implemented in the REDS [38] middleware framework developed in our research group. Our protocols provide REDS with efficient mechanisms to manage content-based routing in dynamic topologies both in a wired and a wireless setting.

Structure of the Thesis

The thesis is structured in four fundamental parts. Part I places the problem into context and describes our overall approach to reconfiguration. Part II address the routing problem and presents a set of optimized solutions for the reconfiguration of the routing information on top of the overlay. Part III addresses the

overlay management problem and presents two novel protocols for the maintenance of a connected tree-based overlay in a peer-to-peer setting. Finally Part IV combines these two layers into a single framework and evaluates their combined performance.

Part I

Problem Definition

CHAPTER
TWO

Background

The increasing availability of network connectivity and of low-cost computing devices has contributed to the development of more and more complex distributed applications interconnecting large numbers of devices over large-scale heterogeneous networks. Web-based applications have become a preferred means to carry out banking transactions, trade stocks, make travel reservations, or to access multimedia content. Peer-to-peer file sharing has become the standard way to gain access to music and software. Overall, a number of applications ranging from telephony to computer gaming benefit from the communication services offered by today's Internet.

In a similar manner, the availability of devices equipped with wireless communication capabilities is fostering the emergence of mobile network scenarios with hosts that are able to roam freely from one location to another. These mobile ad hoc networks play a fundamental role whenever a fixed communication infrastructure cannot be built, or would be too expensive. Disaster recovery solutions adopting these technologies enable quick information dissemination between the operators involved in rescue operations even when fixed communication infrastructures have been rendered unusable by fires, hurricanes, and other events. In a less catastrophic setting, vehicular networks are opening a new frontier for next-generation intelligent transportation systems assisting drivers with traffic information and increasing road safety by allowing vehicles to react to this information automatically. Similarly, low-cost sensing devices equipped with computing and wireless communication capabilities can be deployed over vast areas to form wireless sensor networks enabling retrieval and processing of data in a dynamic and constrained environment.

The need to disseminate information efficiently is a common characteristic of these novel network scenarios and calls for efficient communication paradigms to facilitate the work of application programmers. Traditional client-server communication forces too tight a coupling between communication parties and therefore makes the development of large-scale systems overly complex.

This has led to the development of new communication paradigms that provide a stronger form of decoupling between the components of distributed ap-

plications. Explicit dependencies between components can then be masked by the communication infrastructure, facilitating the ability of the system to scale to large numbers of communicating actors. In addition, by relieving application components from group membership and synchronization issues, the middleware allows application components to communicate with dynamically changing groups of recipients with ease.

2.1 The Publish-Subscribe Paradigm

Among the communication models implementing this strong decoupling, publish-subscribe occupies a prominent position. Both academia and industry have dedicated significant effort to exploring this kind of paradigm, leading to the implementation of a large number of successful middleware systems.

Publish-subscribe enables interaction among a set of applications or application components normally referred to as the system's *clients*. These clients can *publish* event notifications to express the occurrence of relevant events, and they can *subscribe* to notifications regarding the events in which they are interested. Although this communication model was defined in the context of event-notification systems, the contents of notifications can be any kind of messages. For this reason, in the following we will use the terms event, event notification and message interchangeably except for cases where this may lead to confusion and misunderstanding.

Subscriptions can be issued not only for specific events but also for sets of events defined by some type of filters or for patterns describing the occurrence of particular sequences of events. Published events are delivered to all the clients that have issued matching subscriptions by means of an event-notification service or event dispatcher. Subscriptions are stateful operations in that clients continue to receive matching events until they revoke the expressed interests by means of a corresponding *unsubscribe* operation.

The event dispatcher mediates this communication providing the decoupling between publishers and subscribers that characterizes the paradigm. Event notifications are delivered in an anonymous multicast fashion: publishers and subscribers need not have any knowledge about each other. Events can be published regardless of the number of subscribed clients and even regardless of the existence of subscribers. Similarly, subscriptions for a given pattern of events are issued independently of the number of publishers in the systems. Also communication is asynchronous, neither publishers nor subscribers are ever blocked when they issue event notifications or subscriptions, making the paradigm suitable both for large-scale and for mobile environments.

2.2 Publish-Subscribe

The many systems available to date have defined several flavors of publish-subscribe which differ both from a user's and from a middleware designer's perspectives. From the point of view of the user, the first important characteristic is the mechanism employed for selecting and filtering event notifications. According to this

aspect, we can distinguish the basic channel and subject based systems from the more advanced content-based ones.

Topic/Subject Based The simplest incarnation of the publish-subscribe paradigm is based on the notion of topics. Each event notification is associated to one particular topic selected by the publisher, while subscriptions express the interests of clients in receiving notifications belonging to one or more topics.

The simplicity of the topic-based subscription scheme allows for efficient implementations based on multicast routing protocols. Topics can easily be associated to multicast groups so that clients can subscribe to a topic simply by joining the corresponding group. This allows publish-subscribe middleware to be deployed wherever multicast is available. For example, application level multicast protocols can be used in a large-scale peer-to-peer setting, while multicast protocols such as MAODV or ODMRP can be employed to implement topic-based multicast in scenarios like mobile ad hoc networks.

This ease of implementation, however, comes at the cost of flexibility. Topics provide a very limited filtering ability as notifications can only be classified with respect to a single set of topics. The inclusion of multiple classification dimensions is only possible at the cost of a significant proliferation in the number of available topics. This makes it very difficult to match topics with the interests of subscribers, which will generally be forced to issue multiple subscriptions and possibly discard some of the notifications they receive because they are interested only in a subset of them. To make things worse, the set of topics must be defined once for each application. Changing it dynamically requires a new mapping of notifications to topics and forces subscribers to issue new subscriptions to reflect the new topic distribution.

A slight improvement over this flat topic-assignment scheme is provided by systems that organize notifications in a hierarchy of topics, allowing application programmers to group topics using a containment relationship. In this case, subscription filters can be specified using a URL-like format that is somewhat more flexible than the flat subscription scheme available with non-hierarchical topics. Subscribers can, for example, express their interest in a topic and in all its descendant topics using a single subscription.

The use of hierarchical topics also gives application developers the ability to change the set of available topics without needing to rewrite large parts of their applications. For example, they can easily specialize a topic by introducing a new a subtopic for a new type of event. The use of multiple classification dimensions, on the other hand, remains difficult. Adding a new dimension, for example, requires modification of the code for publishing applications so that all events can also be classified according to it.

Content-based Flexibility increases when we consider the more advanced case of content-based systems. In this flavor of publish-subscribe, notifications are published without making any reference to specific topics, subjects or multicast channels. Subscription filters are based on predicates over the content of the event notification. These generally have the form of expressions on the values of a set of

attributes but they can also consist of regular expressions over a textual content or of any other application-defined filtering.

The content-based approach maximizes the decoupling between publishers and subscribers and allows the system to reduce the delivery of uninteresting events to a minimum, avoiding it in all practical cases. These advantages are particularly important in large-scale and dynamic scenarios for at least two reasons. On the one hand, maintaining the common knowledge about topics between publishers and subscribers becomes impractical if not totally infeasible when the sets of topics and clients are both dynamic. On the other hand, delivering unnecessary notification becomes too great a burden in large networks spanning thousands or millions of nodes or in mobile ad hoc networks where high traffic increases the chances of losing network packets.

Needless to say, the greater flexibility of content-based systems comes at the cost of increased implementation challenges. This has motivated significant research to enable the development of scalable content-based publish subscribe systems in a variety of scenarios.

2.3 Publish-Subscribe Extensions

The flexibility of the publish-subscribe paradigm is directly related to the simplicity of the interface it offers to application programmers. In its purest and most basic form, this interface consists only of three basic operations: publish, subscribe, and unsubscribe. Some classes of application, however, require improved middleware semantics which have to be implemented on top of the publish-subscribe paradigm.

To address this requirement, research has been conducted to extend the publish-subscribe interface with new operations. Extensions aim at two primary goals. On the one hand, an improved interface seeks to facilitate the use of the middleware, allowing application programmers to delegate more complex operations to it. On the other hand, it also provides a way for the middleware to perform its functions more efficiently. In the following we review the most important proposed extensions and provide a brief discussion of each.

Advertisements Advertisements [18] are a mechanism to optimize the management of subscriptions. In a standard publish-subscribe system, applications can publish new event notifications simply by invoking the publish operation on the middleware. Advertisements change this behavior by requiring publishers to inform the system that they are going to publish events matching a specific filter.

In addition to matching events against subscriptions, systems implementing this feature match subscriptions against advertisements. If a publisher does not advertise any event matching a particular subscription, then none of its events will need to be checked for matching against the subscription. This allows the publish-subscribe communication middleware to save time and effort when matching events against subscription filters in scenarios where publishers exhibit stable publishing patterns.

Replies Event notifications provide a one-way form of communication with information flowing from publishers to subscribers. Publish-subscribe systems with replies [55, 35] make this communication bidirectional by allowing the receivers of notifications to send responses to publishers using the same communication infrastructure. This allows publish-subscribe systems to be used as a basis for applications that require some form of request-response semantics.

In principle, the ability to reply to notifications could be built on top of publish-subscribe middleware possibly using to point-to-point communication to distribute replies. This, however, requires subscribers to contact publishers directly, violating the anonymity of the publish-subscribe paradigm.

Incorporating replies in the middleware, on the other hand, allows communication to retain the same level of decoupling that characterizes the original paradigm. When replies are sent through the publish-subscribe middleware, subscribers can reply to an event notification without needing to know where the notification originated.

Location Awareness The ability to subscribe to information generated in a specific area or to disseminate information only in a specific region of interest appears as a fairly natural extension to the publish-subscribe communication paradigm [64]. In a fire monitoring application, for example, the control system needs to notify sprinklers and other actuators only if fire is detected inside the room in which they are deployed. In vehicular applications, drivers want to be notified of accidents or traffic jams occurring along the route they intend to follow and not on the opposite side of the city.

Despite the large number of potential applications, however, solutions to address location awareness have only recently been proposed in the context of publish-subscribe middleware [7, 41]. This is likely due to the fact that content-based publish-subscribe systems allow locations to be treated in the same way as any other attribute allowing clients to constrain it in their subscriptions. Nevertheless while this approach allows location to be evaluated at subscription time, it cannot allow publishers to direct the distribution of event notifications only to certain areas, another interesting feature that is often required in location-aware systems.

Location aware publish-subscribe addresses this issue by incorporating the concept of location into the middleware interface. When publishing an event notification, clients can specify the physical regions for which the notification is relevant. Similarly, when subscribing, they can express their interest in notifications regarding or generated in specific areas.

In addition to providing more powerful semantics than traditional systems, location-aware publish-subscribe facilitates application development by providing a cleaner interface to the programmer. Moreover the explicit representation of location allows the middleware to optimize the processing and propagation of notifications and subscriptions by constraining their diffusion to the geographical areas specified by publishers and subscribers.

2.4 Implementation of Publish-Subscribe

Up to this point we have focused on the services offered by publish-subscribe middleware to the application programmer. In this section we continue our analysis from a different perspective, namely that of the middleware designer. Specifically, we shift our interest to the middleware component that enables the decoupled communication style typical of this paradigm: the event dispatcher.

2.4.1 Architecture of the Event Dispatcher

One of the major design choices characterizing available solutions for delivering event notification in publish-subscribe systems is the architecture of the event dispatching system.

The simplest solution exploited by the earliest publish-subscribe implementations is that of a centralized event dispatching component. Clients, i.e. both publishers and subscribers, are connected to a dispatching server responsible for collecting and remembering subscriptions. When a notification is published, the event dispatcher matches it against its subscription table and forwards it to all the clients with a matching subscription.

Clearly, a centralized solution exhibits problems when scaling the system to large numbers of clients distributed over a wide area network. For this reason, a number of distributed dispatching solutions have been proposed, both in research and in industrial-strength products.

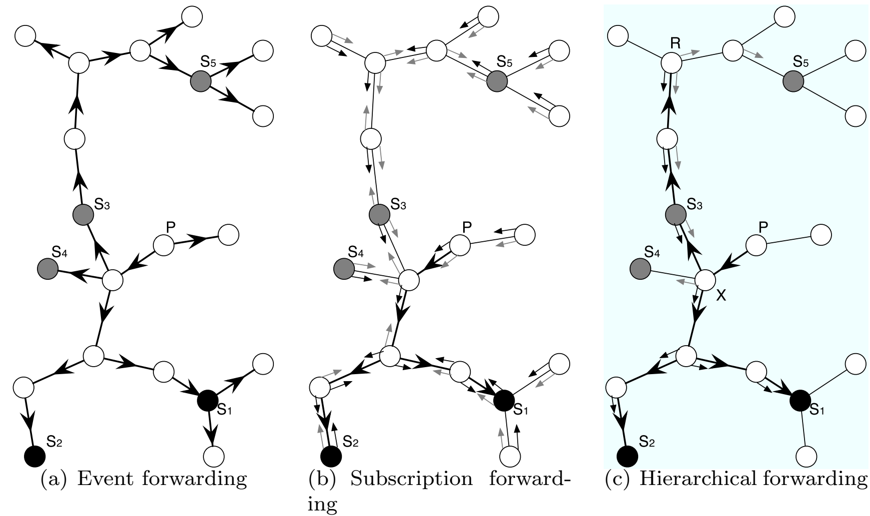
A distributed event dispatching system consists of a network of dispatchers, often referred to as brokers, which cooperate to collect subscriptions and to deliver event notifications to interested subscribers. Clients connect to the distributed event dispatcher by connecting to one of its brokers.

A key aspect in the design of a distributed event dispatcher is the topology of the *overlay* network connecting its brokers. A number of different topologies have been described in the literature, ranging from acyclic trees to more connected mesh overlays.

Tree-Based Routing

The most common way to interconnect the brokers in a distributed event dispatcher is to use an acyclic graph, that is a tree-like topology. The reason is that an acyclic structure simplifies routing by naturally preventing cycles in the propagation of notifications. Moreover, the tree can be computed in advance to minimize the cost of propagating messages. In pure topic-based systems, each event is associated to one and only one topic. As a result, topic-based event dispatchers usually maintain multiple distribution trees, one for each topic, as is done in multicast routing protocols.

In content-based systems, on the other hand, events cannot be associated to any specific topic. For this reason, content-based dispatchers almost always exploit a single distribution tree (or graph) to propagate all event notifications. A number of different message-routing strategies can be used for this propagation: the most relevant being subscription forwarding, event forwarding, hierarchical forwarding, and advertisement forwarding [18].



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

Figure 2.1: Publish-subscribe routing schemes. Circles denote dispatchers. Filled circles represent dispatchers subscribed to a given "color" pattern. Colored 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.

Event forwarding In the simplest routing strategy, events received by a dispatcher from one of its clients are simply broadcast along the tree connecting all the dispatchers. The information about a client's subscriptions, on the other hand, is never communicated and is stored locally to the dispatcher the client is attached to. This information, stored in the dispatcher's *subscription table*, is checked whenever an event is received (forwarded by a neighboring dispatcher or published by another of the attached clients), to determine whether any of the attached clients should receive a copy of the event. Figure 2.1(a) illustrates the concept. Two dispatchers, S_1 and S_2, are subscribed to the same pattern, represented by the black color, while dispatchers S_3, S_4, and S_5 are subscribed to a "gray" pattern[1]. When the dispatcher P publishes an event matching the black pattern, the corresponding message is forwarded along the thick, directed lines shown in the figure, i.e., to all the other dispatchers, including the intended receivers S_1 and S_2.

Note that in reality only clients are subscribers and/or publishers of events. However, with some stretch of terminology we say that a dispatcher is a subscriber (publisher) if at least one of its clients is a subscriber (publisher). For this reason, clients are omitted from the graphical representations in Figure 2.1.

[1]In practice, event patterns can be sophisticated expressions, e.g., involving regular expressions as in ⟨`"Distributed Sys*" OR "Soft?are", 52.0`⟩.

Subscription Forwarding Event forwarding inevitably results in high event traffic as all events are sent to all dispatchers, regardless of the presence of receiving clients. Subscription forwarding limits this overhead by spreading knowledge about subscriptions beyond the dispatcher the subscribing client is directly attached to. Specifically, when a client issues a subscription for a given event pattern, not only is the pattern inserted into the subscription table of the dispatcher the client is attached to, together with the identifier of the subscriber (as in event forwarding), but the subscription message is also forwarded to all the neighboring dispatchers. During this propagation, each dispatcher behaves as a subscriber with respect to its neighbors, i.e., it records the event pattern in its subscription table and re-forwards the subscription to its neighbors, except the one that sent it. This scheme is usually optimized by avoiding subscription forwarding of the same pattern in the same direction. 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 2.1(b) illustrates the concept graphically.

Hierarchical forwarding The hierarchical forwarding strategy strikes a balance between the two aforementioned schemes. The strategy assumes that dispatchers are organized in a rooted tree. Subscription messages are always forwarded towards the root of the tree but they are never propagated "downstream". Events are similarly always propagated "upstream" towards the root. Nevertheless, at each dispatcher, including the root, events can also propagate "downstream", if a matching subscription dictates so. This is shown in Figure 2.1(c), where the dispatcher R acts as the root. Events published by P are forwarded up to the root R, but are also steered downstream by the subscriptions issued by the various subscribers. It is worth noting that an event from P to S_1 and S_2 will not pass through the root before being redirected to the subscribers. Instead, at the branch point in the tree, X in figure, the event is copied and sent both downstream to S_1 and S_2 and upstream to the root.

Advertisement-based routing In Section 2.3, we discussed advertisements as a possible extension to the interface of publish-subscribe middleware. From the point of view of routing protocols, this extension enables a restriction in the propagation of subscriptions in the same way as subscriptions enable a restriction in the propagation of event notifications. As a result, advertisement can, in principle, be managed using any of the above strategies. In practice however, a strategy that keeps advertisements local as event forwarding does for subscriptions corresponds to a strategy that does not use advertisements; this leaves the choice between hierarchical advertisements and advertisement forwarding. To the best of our knowledge, only advertisement forwarding has been implemented in available systems [18].

Subscription and Advertisement Coverage A useful optimization that can be implemented in the aforementioned protocols is based on a relationship, *covering*, defined between the patterns (or filters) contained in subscriptions and

advertisements. In very simple terms, a filter (or pattern) covers another if it matches all the notifications matched by the latter.

Routing protocols can take advantage from this relationship to limit the propagation of subscriptions (and advertisements) [18]. Dispatchers propagate subscriptions only for patterns that are not completely covered by previously issued subscriptions. If a subscription for "temperature values over 30 degrees" has been propagated, there is no need to propagate a new subscription for "temperatures over 45 degrees". This relation between patterns allows the system to quench the propagation of subscriptions but at the same time it complicates the processing of unsubscriptions issued by different clients.

Non Tree-Based Routing

Organizing a distributed message dispatcher in an acyclic topology is not the only solution available in the literature. Recent years have seen the development of a number of routing schemes for content-based publish-subscribe that are not based on a single tree topology. The idea of a dispatcher operating over a general graph topology is described in [18], and the same authors have recently proposed two different approaches [19, 54] that route messages using multiple data distribution trees.

The approach in [19] exploits a content-based routing protocol resembling a subscription forwarding strategy with subscription coverage, combined with a broadcast layer that builds per-source distribution trees. Routing information is updated with a combination of a receiver-based "push" mechanism and a source-based "pull" approach. In the first, receivers periodically propagate receiver advertisements specifying their "subscription information". In the second, message sources periodically send Update Request messages to update routing information on their data distribution tree. The refresh mechanism allows the protocol to address some topological changes.

In [54], the same authors describe a protocol for content-based routing in sensor network. The protocol builds forwarding trees rooted at message receivers and also maintains alternate paths for routing around network failures. Trees are refreshed periodically, however the protocol also implements a reactive approach that attempts to repair trees when failures occur.

A different way to exploit multiple routing trees is instead used in Kyra [16]. Kyra builds several routing trees, each responsible for forwarding a subset of events. Dispatchers are then organized in a set of server cliques and each dispatcher acts a proxy for a subset of the subscriptions generated in its clique. The work in [88] instead trades off transport cost for reliability and forwards messages on a mesh topology in which each node is connected to n distinct parents. Finally, Speccast [80] uses multiple trees to implement a communication paradigm with predicates associated to messages. Although not directly part of the publish-subscribe literature, this work presents a general routing service that can be exploited in publish-subscribe middleware.

As we describe in Chapter 8, topologies like multiple trees, meshes, and even structure-less systems [34, 7] are often employed to add reliability and load balancing to publish-subscribe middleware. However in this thesis, we show that

content-based publish-subscribe middleware can operate reliably on a tree overlay that is reconfigured in the presence of node and link failures, thus achieving greater efficiency with respect to systems based on more connected topologies.

2.4.2 Matching Events against Subscriptions

A second aspect in the implementation of a publish-subscribe dispatching system is the mechanism used for matching events against subscriptions. While the problem is trivial in the context of topic-based systems, it becomes a major concern in the design of content-based ones.

In such systems, the trivial solution that tests each event against each subscription often becomes inapplicable for its poor performance. Research has therefore identified several approaches based on various forms of indexing and decision diagrams. These techniques can be grouped in two main categories according to how they iterate through the predicates and constraints.

The first category uses the data contained in event notifications to move through the constraints defined by the whole set of subscriptions at a given broker. This is accomplished by means of matching trees [2, 51] or binary decision diagrams [15]. More specifically, the matching tree or decision diagram is built using the various predicates contained in subscriptions. Each node in a given level in the tree represents the same predicate and has at most three successors corresponding to the values “true”, “false” and “don’t care”. Subscriptions correspond to the sequence of predicates obtained in a path from the root to a leaf. A given event is processed starting from the root node and continuing along all the branches whose predicates are satisfied by the event.

The second category of approaches moves through the predicates contained in the event consulting the constraints defined by subscriptions [96, 74, 20, 44]. Subscriptions are decomposed in elementary constraints. An event is matched by iterating over its properties: for each event property the matching algorithm records which subscriptions have a corresponding matching constraint. Once it has iterated over all event properties the algorithm returns to subscriptions. Matching subscriptions are those for which the number of matching constraints equals the total number of constraints in the subscription.

2.5 Existing Publish Subscribe Middleware

The body of research in the context of publish-subscribe has led to the development of a number of middleware platforms both in academia and industry. In the following, we provide a brief description of some of the major systems highlighting their main characteristics.

TIB/Rendezvous TIB/Rendezvous [92] is a commercial publish-subscribe infrastructure developed by TIBCO. TIB/Rendezvous events comprise a set of typed data fields, in a record-based fashion, while the subscription language is subject-based. Subscription can filter events based on the value of a special field which acts as the subject.

The dispatcher exploits a three-level hierarchical architecture. Each network node running a client must also run a TIB/Rendezvous daemon, responsible for filtering the events for the clients on that node. The daemons communicate by means of broadcast messages within the same subnet. Inter-subnet communication is achieved through two levels of routing daemons: *subnet routing daemons* and *wide-area routing daemons*.

SIENA SIENA [18] is a wide area event notification service developed at Politecnico di Milano and the University of Colorado at Boulder. Its main objective is to maximize the expressiveness of the subscription mechanism without sacrificing scalability. SIENA event notifications are defined as sets of attributes characterized by a type, a name, and a value. Its subscription language is content-based and enables the definition of filters on the content of a single notifications or of patterns of filters matching specific event combinations. The event dispatcher exploits a distributed architecture and routing is performed using a hierarchical, a subscription forwarding, or an advertisement forwarding strategy. In all cases SIENA employs covering relationships between patterns and filters to minimize the propagation of subscriptions and advertisements.

LeSubscribe Le Subscribe [45] is a content-based publish-subscribe system developed designed and developed at INRIA. The project aims to support reactive applications consisting of large numbers of distributed components. Different from our approach, Le Subscribe's research group focuses primarily on the filtering problem to enable efficient matching of events with a large number of subscriptions.

JEDI JEDI [35] (Java Event-based Distributed Infrastructure) is an object-oriented event-based infrastructure developed at Politecnico di Milano. JEDI is a tuple-based notification system: its events are defined as ordered sets of strings, where the first string is the *event name* and the remaining strings are the *event parameters*. Subscriptions contain a set of strings which may optionally contain wildcards, and they match the events with the specified number of fields and corresponding values for the strings in their tuples. JEDI provides both a centralised and a distributed implementation of the event dispatcher, the latter being based on a hierarchical forwarding strategy.

Hermes Hermes [77, 78] is a scalable and reconfigurable publish-subscribe middleware platform that uses peer-to-peer techniques to build and maintain its overlay routing network. Hermes provides a slightly limited form of content-based routing, termed "type and attribute based". Each message type is associated to a rendezvous point, which takes the same role as the core node in core-based tree multicast [9].

Gryphon Gryphon [90] is the result of a research project by IBM aiming to build a robust publish-subscribe message broker. At its core is a distributed filtering algorithm based on parallel search trees to allow brokers to determine

where to route messages. The architecture of the event dispatcher is distributed over a tree topology and replication is used to improve the system's reliability.

Xnet XNet [25] is a content-based network developed at Eurecom. The project focuses on the design of a scalable and reliable system for the distribution of structured XML content to large populations of consumers. Its routing protocol [24] exploit aggregation of subscriptions to limit the size of routing tables without increasing notification traffic.

REBECA Rebeca [47] is a notification service incorporating several routing strategies based on subscription leasing on top of a rooted tree of brokers. The leasing approach allows the system to self stabilize without any special mechanism but it requires consumers to renew their leases at regular intervals. This approach increases the overhead of the system and may lead to consumers receiving notifications in which they are not interested any more because subscriptions remain valid until the corresponding leases have expired.

Elvin Elvin [86] is a publish-subscribe notification service developed at the Distributed Systems Technology Centre. Elvin provides a content-based subscription language to filter notifications consisting of multiple fields of several data types. The system can be distributed over a wide-area network through by interconnecting clusters of brokers. Scalability is facilitated by *notification-quenching*, an approach allowing publishers to generate only the events to which at least one subscriber is interested.

CHAPTER
THREE

Motivation and Approach

One of the prime characteristics of content-based publish-subscribe is the sharp decoupling it introduces between communicating parties. Applications can exchange messages without worrying about recipient lists or predefined destination groups. The middleware simply delivers messages to their correct recipients based on their content and on the interests expressed by communicating parties. This makes the paradigm naturally suited to the development of applications in scenarios where communication parties change their interests dynamically as the application evolves. The middleware hides the complexity of communication aspects from application components therefore facilitating development and preventing potential implementation errors.

The dynamic behavior of application components however is only one of the aspects of modern distributed applications. The increasing availability of low-cost network connectivity has led to new large-scale collaborative applications spanning large numbers of user that can connect and disconnect at any time. Similarly, the widespread diffusion of ubiquitous computing resources has led to the need to support communities of nomadic users able to roam between different network domains.

Despite its extreme flexibility, currently available publish-subscribe middleware offers only limited support for these new dynamic scenarios. Some middleware systems allow client applications to join and leave the network dynamically as they move from one broker to another in different logical or physical locations. However, the most common underlying model is still that of a platform based on a stable network of servers whose disconnection is a rare event almost always due to failures. The event dispatcher, albeit distributed, is usually deployed on a set of dedicated machines. As a result, even systems that support dynamic dispatching topologies, do so only with basic protocols, characterized by a significant reconfiguration cost.

This calls for new middleware platforms able to provide application developers with efficient communication in dynamic scenarios, an essential tool for the development of complex large-scale distributed applications. This thesis addresses this issue, with a middleware a solution for content-based publish-subscribe com-

munication in large-scale networks characterized by frequent disconnections or by the presence of mobility. In the rest of this chapter, we introduce our approach and outline its major characteristics. Section 3.1 describes the two main network scenarios we target in our work. Section 3.2 briefly discusses some existing solutions for publish-subscribe in dynamic networks. Section 3.3 presents the overall architecture of our middleware solutions and finally Section 3.4 closes this first part of the thesis with some concluding remarks.

3.1 Targeting Dynamic Scenarios

The design of solutions for middleware in dynamic network environments covers a broad spectrum of scenarios ranging from small scale corporate networks to large scale heterogeneous systems; from sensor networks characterized by limited computing and battery power to highly mobile ad hoc networks. Corporate networks may be subject to occasional administrative and organizational changes. Nodes in a large-scale heterogeneous systems may be characterized by the most diverse and unpredictable connectivity patterns. Battery management policies may force sensor nodes to join and leave the network at frequent intervals, while mobility may cause frequent topology changes in ad hoc wireless networks.

In this thesis, we develop a middleware solution to address the challenges posed by these dynamic environments. In doing this we focus primarily on large-scale peer-to-peer networks. Nevertheless, we also present solutions that enable our middleware to operate in the context of mobile ad hoc networks. The rest of this section provides a brief description of these two main scenarios.

Large-scale peer-to-peer networks The last few years have seen a growing interest in large-scale distributed applications involving large numbers of powerful edge-nodes and multiple service providers. The availability of computing power and broad-band network connectivity has enabled the development of decentralized applications in which communication and coordination are managed in a completely decentralized fashion without relying on stable networks of servers.

On the surface this kind of applications raised public interests as a powerful solution to prevent centralized control over the distribution of information, music, and other types of digital content. However, these decentralized architectures have their major advantages in terms of scalability and reliability. First, they do not rely on computational entities that constitute single points of failures. Second, they are inherently self-organizing and do not require the presence of a central administration authority.

In this work we show that these advantages can be achieved in publish-subscribe middleware. Specifically we build a decentralized middleware infrastructure and allow it to reconfigure its operation as a result of changes in the underlying network. In the case of large-scale peer-to-peer networks these changes usually define a model in which network nodes can join and leave the network at arbitrary times without explicit announcements. This type of changes cannot be handled transparently by the underlying network layers and must be handled explicitly by the middleware.

Mobile ad hoc networks Mobile ad hoc wireless networks (MANET) are self-configuring sets of mobile nodes that communicate through a wireless medium in the absence of any fixed infrastructure. The movement of hosts causes network topology to change unpredictably and the absence of an infrastructure forces the use of decentralized routing protocols.

In a MANET environment nodes communicate via broadcast communication over a wireless medium. In the absence of multi-hop routing protocols, a message reaches all network nodes that are within communication range of the sending node. Moreover, messages can specify the destination address of a node in this range, allowing nodes to discard messages not directed to them. This type of communication is enabled by the 802.11 MAC-level protocol, which is the one we assume in the deployment of our middleware in MANET scenarios.

This brief description of the MANET environment highlights a fundamental difference with respect to the peer-to-peer scenario described above. While it is perfectly possible for a MANET node to disconnect, by switching off its radio at any time, the most frequent type of *change* in the structure of the network results from the ability of nodes to roam from one physical location to another. This implies that a given network node can communicate directly with a subset of other nodes that changes dynamically over time.

3.2 Existing Solutions

Although each of the scenarios we just described is characterized by very specific characteristics, the common theme is that the middleware can no longer assume to be based on a stable network topology. It is not simply a matter of client applications roaming from one broker node to another: broker nodes themselves need to rearrange their interconnections as other broker nodes appear, move in or out of communication range, or disconnect from the network.

This has motivated the emergence of several solutions for publish subscribe in dynamic environments. Small-scale wireless networks characterized by a high degree of mobility have motivated the emergence of dissemination protocols [34, 7] that do not rely on stable routing information and replace it with a combination of soft-state information and probabilistic message forwarding. These approaches achieve good performance in wireless networks characterized by very high mobility patterns, but they are inherently inconvenient whenever deterministic routing information can be maintained. This motivates the use of deterministic routing whenever possible, leaving alternative approaches to those cases in which the scenario does not allow a routing infrastructure to be maintained and used correctly.

Other efforts have instead addressed the problem with resilient deterministic routing approaches. These most often exploit graph overlay topologies in which senders and receivers are connected by means of multiple communication paths [25]. The presence of multiple paths allows the system to continue to operate even in the presence of communication failures, but results in unnecessary overhead when operating in stable network conditions. For this reason most systems build data distribution paths over these graph topology to reduce communication overhead [19]. These distribution paths however must be refreshed periodically to account for changes in the topology of the underlying network. The refresh

period must be short enough to allow the system to adapt to topology changes but also long enough to avoid consuming most bandwidth for refreshing routing information.

In this thesis, we take a different approach and update routing information reactively in response to topology changes. Moreover, we do this by directly maintaining a tree overlay topology on top of a dynamic network infrastructure. Our results show that the approach is not only able to achieve very good performance in large-scale wired networks, but also that it is a valid solution for the development of middleware for mobile ad hoc networks.

3.3 Reconfigurable Tree-Based Publish-Subscribe

The goal of the work presented in this thesis can be summarized in terms of a fundamental goal: *build a scalable middleware infrastructure to enable content-based publish-subscribe communication in dynamic networks ranging from large-scale peer-to-peer to mobile ad hoc networks.* With this fundamental goal in mind, we now describe the main characteristics of our middleware solution. We begin by stating our two core design choices. Then, we describe our high-level middleware architecture and finally we discuss its ability to operate in multiple scenarios in a uniform way.

3.3.1 Key Design Choices

The first of our design choices follows from our target application scenarios: peer-to-peer and mobile ad hoc networks. As we discussed above, both scenarios benefit from decentralized middleware architectures. This allows the middleware to achieve increased reliability by eliminating centralized points of failure and by avoiding reliance on statically defined subsets of machines that act as coordinators for the operation of the middleware.

To achieve both of these goals we define our architecture as one in which all, or at least most, network nodes actively participate in routing messages. That is *all, or most, network nodes act as brokers in a distributed message dispatcher.* This is almost a forced choice in a MANET environment [57] where relying on a predefined set of brokers is not possible as all nodes may move arbitrarily and cause brokers to become unreachable. However, it is also a reasonable choice in peer-to-peer networks in that our goal is to build a system in which we make no assumptions on the connection time of any network node. In both cases, it would in principle be possible to elect a subset of brokers dynamically among network nodes, but this solution would likely result in additional overhead for maintaining dynamic membership information for the subset of nodes that act as brokers.

The second design choice we face in the design of our middleware is that of the message routing strategy. As we discussed in Chapter 2, a very common approach to routing in content-based publish-subscribe systems is to arrange the set of brokers in an unrooted tree topology. Routes for message propagation are then laid out on top of this topology to enable the delivery of messages to all interested subscribers. Nevertheless, the use of a single data-distribution tree is not the only viable option. As we discussed above, a significant portion of recent

work on communication middleware for dynamic networks replaces tree topologies with more connected mesh overlays on the premise that trees are inherently fragile and thus not suited to scenarios where connectivity changes dynamically.

In our work we confront this assumption and push the limits of tree-based message routing. In particular we show that not only is it applicable in highly dynamic scenarios but also it provides very good performance both in terms of reliability, i.e. percentage of delivered messages, and efficiency of communication. To achieve this, we extend existing tree-based message routing with a notion of reconfiguration.

3.3.2 A Reconfigurable Middleware Architecture

As discussed in Chapter 2, routing in content-based publish-subscribe systems is managed by a network of cooperating dispatchers (or brokers). In a stable network topology, the interconnections between these brokers are established statically at deployment time. Only routing information is established dynamically as new subscriptions, unsubscriptions, and possibly advertisements are issued. When the network topology is dynamic, on the other hand, the routing infrastructure must be *reconfigured as the system evolves, that is it must adapt to the changes in the underlying physical network without interrupting the normal operation of the system.*

Our middleware solution addresses this reconfiguration of the routing infrastructure in content-based systems as the sum of three subproblems, namely:

1. maintaining the overlay network interconnecting message brokers, repairing disconnections caused by failures without creating loops;

2. reconciling the subscription information held by each broker 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 this reconfiguration process.

To address these subproblems we propose the middleware architecture depicted in Figure 3.1. It consists of three fundamental layers: the overlay, the routing layer and the event-recovery layer. Decomposing the middleware into these layers facilitates the work of designers as each of each middleware component can be reused in different situations and scenarios. For example, a middleware system can exploit the same routing component in peer-to-peer and in mobile ad hoc networks by combining it with the appropriate overlay layer. In the rest of this thesis we present a set of protocols for the overlay and routing layers. The protocols are evaluated by means of extensive simulations and are implemented[1] within the REDS middleware framework [38].

Figure 3.1 shows how the protocols occupy the layers of the architecture. Each protocols is represented by a box with thick contour. Those with a white background are protocols available in the literature that we use either as building

[1]With thanks to Alessandro Monguzzi for his implementation effort.

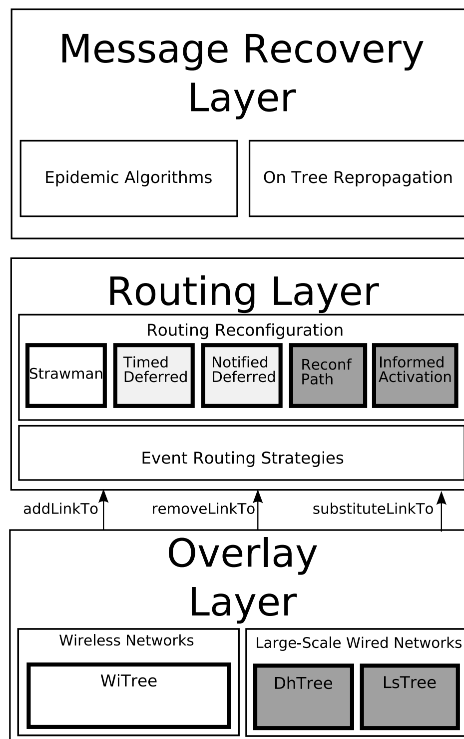


Figure 3.1: The three layers of reconfigurable publish-subscribe middleware.

blocks for our middleware or as baselines for comparison. Boxes with shaded background represent a contribution of this thesis in either of two ways. A light background indicates that our contribution is in the simulation study, while a darker background indicates that the contribution is both in the design of the protocol and in the simulation study.

In the rest of this section we complete our discussion of the middleware architecture with a description of the overlay, routing and event recovery layer.

Overlay Layer The overlay layer sits at the bottom of our overall architecture and constitutes the interface of our middleware with the underlying network and operating system. According to the above design choices, this layer provides an overlay network abstraction that consists of an unrooted tree topology. The overlay is built and maintained with mechanisms determined by the specific deployment scenario. In a wired setting, the overlay is formed as a subset of the complete-graph abstraction provided by the underlying IP-routing mechanism. In a wireless network, on the other hand, it is constrained by the communication range of wireless hosts and does not rely on any multi-hop routing protocols. The use of such protocols would in fact duplicate the maintenance effort already required at the middleware routing layer.

The overlay layer manages all implementation aspects deriving from specific network environments. Specifically it presents a uniform interface to the routing layer and thus allows it to be moved from one scenario to another simply by changing the underlying overlay manager. The interface between routing and overlay is described in detail in Section 3.3.3.

The separation of the overlay from the routing layer also allows the overlay to find applicability outside the domain of content-based publish-subscribe middleware. For example, most subject-based publish-subscribe systems generally route messages using multiple overlay trees for each subject. Similarly, application level multicast protocols adopt a set of overlay trees rooted at the multicast source. Both of these types of systems can exploit the overlay managers presented in this thesis for managing their communication infrastructure.

Part III of this thesis presents two overlay protocols for large-scale wired scenarios. The first, described in Chapter 10, builds the overlay directly on top of the TCP/IP network layer. The second, described in Chapter 11, exploits a Distributed Hash Table to maintain an overlay conforming to the characteristics of a reference tree topology. In addition in Chapter 14, we briefly describe WiTree, an overlay manager for wireless network by others in our research group [67]. These solutions are finally combined with the routing layer in the evaluation presented in Part IV.

Routing Layer Built on top of the overlay, the routing layer is the key component of content-based publish-subscribe middleware. A routing layer is responsible for the correct distribution of messages using the routing strategy that mostly fits the specific application scenario. As we pointed out above, the architecture presented in this thesis is based on a subscription forwarding routing strategy, although in Chapter 7 and in Part IV, we briefly consider other routing strategies as terms of comparison.

The novel aspect in the routing layer presented in this thesis is its ability to adapt to changes in the underlying overlay topology. A dynamic overlay manager provides an overlay topology that changes according to the requirements of the underlying physical network. Maintaining the same overlay topology despite changes in the physical network is not only inefficient but also impossible in cases in which the set of broker nodes changes dynamically. The routing layer responds to these changes by reconfiguring its routing information while the system continues to operate. This part of the reconfiguration problem is addressed in Part II where we describe several routing reconfiguration protocols characterized by different trade-offs between requirements and performance.

Event-Recovery Layer The top-level component of our architecture is the event-recovery layer. This layers aims to make the reconfigurations of the message dispatching system totally transparent to the applications built on top of it. The modifications in message routes determined by dynamic network topologies necessarily cause some messages to be lost while the overlay and routing layers perform their reconfigurations. The event recovery layer aims to reduce this effect by caching and re-propagating messages as needed. Precise details about this process are outside the scope of this thesis and have been addressed by parallel work in our research group [33].

3.3.3 Addressing Different Failure Models

The architecture outlined in the previous section allows publish subscribe middleware to operate in dynamic network environments by reacting to changes in the underlying physical network. This suggests that the reconfiguration mechanisms employed by the middleware must depend on the characteristics of the specific application scenario and in particular on the *failure* model[2] it defines. In this section, we review the failure models associated with our target network scenarios and discussed how they are managed by our middleware architecture.

Underlying Failure Models: Nodes and Links

As we already pointed out the the work presented in this thesis targets two main application scenarios: large-scale peer-to-peer and mobile ad hoc networks. The characteristics of these two scenarios are inherently different. In peer-to-peer, the dynamic nature of the network results mainly from the ability of hosts to join and leave at unpredictable moments. In MANETs, on the other hand, topology changes are almost always the result of mobility. The ability of hosts to roam from one physical location to another determines the appearance and disappearance of communication channels between them, and while a host may decide to leave the network and disconnect, this is not the most frequent case.

The differences between these scenarios are mostly evident at the overlay layer. An overlay for mobile ad hoc networks inevitably uses different mechanisms from one designed for a large-scale peer-to-peer environment. However, our layered

[2]The ability to reconfigure is reminiscent of a form of fault-tolerance except for the fact that in the scenarios we target network changes constitute the norm rather than the exception.

middleware architecture still allows us to treat different scenarios in a uniform way. The overlay layer is in fact able to mask the characteristics of the specific scenario providing a high-level topology abstraction to the routing layer. Changes at the network level are translated into changes in this topology abstraction. This provides a uniform network model to the protocols at the routing layer and makes them independent of the specific failure model. This allows our middleware solution to migrate easily from one environment to the other simply by choosing the most appropriate overlay layer. In the following we define the overlay topology abstraction by describing the interface between overlay and routing layers.

Abstract Failure Model: Interface between Routing and Overlay

The core characteristic of the interface provided by the overlay layer is that each node's routing layer is made aware of the identity of its neighboring nodes in the overlay. Neighborhood information satisfies a basic symmetry property. If a node A recognizes another node B as a neighbor then B must also recognizes A as a neighbor. Similarly if A loses its connection to B, then B also loses its connection to A. On the other hand, a situation in which the same node A appears disconnected to a node B but is recognized as a working neighbor by another node C is perfectly acceptable.

Each middleware node is only aware of the existence of overlay links between its neighbors. This means that both the peer-to-peer and the MANET scenarios can be handled in a uniform way by the routing layer. The disconnection or the joining of a node is in fact visible only to its overlay neighbors and manifests itself as the appearance or the disappearance of a link to each of them.

The routing layer exploits the information about the identity of its overlay neighbors to exchange subscriptions, unsubscriptions, and event notifications with each of them as specified by its routing strategy. Whenever the physical topology changes, the overlay layer reacts and possibly modifies the overlay topology. When this happens, it notifies the routing layer of which links to neighboring nodes have appeared or disappeared. These notifications allow the routing layer to maintain its routing information consistent with the overlay.

The notification of the appearance and disappearance of links comes in several flavors. In all cases, the overlay layer informs a node's routing layer about the acquisition or the loss of a link to neighbor. However, different routing reconfiguration protocols require different information from the overlay layer. For example, some protocols require that nodes that lose a link to neighbor be informed about other new links that have been inserted in the overlay. This is represented in Figure 3.1 by the three calls made by the overlay to the routing layer. The *addLinkTo* call informs the routing layer of the existence of a new neighbor; the *removeLinkTo* informs it of the loss of a neighbor; and finally *substituteLinkTo* informs it that the link to a neighbor has been lost and will be replaced by another link specified as a parameter. A detailed discussion of these calls is provided in Part III along with a description of the corresponding routing reconfiguration protocols.

3.4 Concluding Remarks

This chapter presented a high level description of the middleware solution presented in this thesis. Its main features are the ability to operate in different types of dynamic network environments and to reconfigure its operation in response to the changes topology of the underlying physical network. In the rest of the thesis we detail the characteristics of the components of this architecture. Part II presents our routing reconfiguration protocols and discusses the requirements they pose on the overlay management layer. Part III addresses overlay management and describes two protocol for maintaining a tree overlay in peer-to-peer networks. Finally Part IV integrates these two layers evaluating their performance by means of simulations.

Part II

Routing Layer

CHAPTER
FOUR

Routing Requirements and Goals

In the previous chapters we described the problem of reconfiguration in content-based publish-subscribe systems and outlined how it can be addressed with a middleware solution consisting of the combination of three layers of protocols. In this part of the thesis we address the middle layer in this solution and present the work in [40], describing a set of protocols for the reconfiguration of event routing. Our analysis of the routing reconfiguration layer 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 may be undermined.

The core of the routing layer is constituted by the techniques described in Chapter 2 to address event dispatching in content-based publish-subscribe. In this part of the thesis, we augment these techniques with mechanisms for the reconfiguration of routing information when the underlying topology changes as a result of a node or a link failure, thus extending their applicability from stable to dynamic and mobile networks. Among all the strategies presented, we base our reasoning on subscription forwarding as it is both the most widely used and the one offering the highest performance in scenarios characterized by significant event load. Subscription forwarding builds the routes for event distribution by propagating subscriptions and unsubscriptions issued by clients. The purpose of reconfiguration protocols is to modify these routes effectively to enable the design of scalable and efficient middleware in dynamic environments.

The basic requirement for these protocols is the ability to restore a correct routing of event notifications after an arbitrary sequence of reconfigurations. This means that events should neither be lost nor propagated unnecessarily unless for the limited and ideally very short period of time needed by the reconfiguration process.

In addition to maintaining this basic correctness requirement, reconfiguration

protocols should strive to maintain some simple desirable properties. First, routes should be modified with as little communication effort as possible to minimize the impact of reconfigurations on the overall performance of the middleware. Second, the system should return to its stable behavior as quickly as possible: this allows it to withstand higher rates of reconfiguration without incurring in additional overhead and event loss caused by interfering reconfigurations. Third, event loss should be kept to a minimum even during reconfigurations to facilitate the job of the event-recovery layer.

In the scenarios we target, large-scale wired and mobile ad hoc networks, sources of reconfiguration can be modelled as the appearance or disappearance of a node or a link respectively. Nevertheless, as we pointed out in Section 3.3.3, both cases manifest themselves as the acquisition or loss of a link to a neighboring broker. For this reason, in this part of our work, we consider the acquisition and loss of links as our basic building blocks. This allows us to focus on the essence of the reconfiguration problem and unveil its fundamental characteristics and tradeoffs. In Part IV, we return to this issue and show how our protocols can also be successfully employed when brokers leave or join the network unpredictably.

The description of the protocols in the following chapters assumes 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.

Finally, while the details about how the tree maintenance sub-system operates are given in Part III, each routing reconfiguration protocol has slightly different requirements on its interface and functionalities. We will analyze them as part of the description of the different protocols in Chapter 6. Here we observe that at a minimum the tree maintenance sub-system must notify 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.

Our analysis of the routing reconfiguration problem unfolds as follows. Chapter 5 presents a basic solution found in the literature and suggests possible strategies for improvements; Chapter 6 describes two basic improved protocols as well as our new contributions. Chapter 7 provides a detailed analysis of the protocols by means of simulation; and finally, Chapter 8 describes some related and alternative approaches to the routing problem in content-based publish-subscribe.

CHAPTER
FIVE

Baseline and Observations

In this chapter, we initiate our description of routing reconfiguration and set the basis for the optimized solutions presented in the following chapters. Our starting point is the Strawman approach, a basic solution found in the literature [18, 98], which is based on the subscription forwarding strategy described in Section 2.4.1. The Strawman protocol provides an ideal starting point for our investigation for three main reasons. First, it serves as a base for understanding reconfiguration and how it can be improved. Second, it serves as a point of comparison for demonstrating the improvements achieved by the protocols we consider and particularly by those introduced in this thesis. Finally, it uses only the normal operations available in the subscription forwarding strategy, allowing us to introduce, in a simple context, the notation that we use later for describing more complex protocols.

5.1 Baseline: Strawman Protocol

The operation of the Strawman protocol is based on two key observations. First, when a link breaks, messages can no longer be sent across it and therefore all subscriptions received previously along that link using subscription forwarding are useless and should be removed. Second, when a dispatcher is notified that a new link is to be added, it must ensure that any events that are of interest and are generated by the other end-point (or by any dispatcher in its sub-tree) are forwarded properly across the new link. The key to the Strawman protocol is that the operations above can be accomplished entirely with normal subscription and unsubscription messages.

Protocol description. When a dispatcher is notified that a link to one of its neighbors is broken, it behaves as if it received unsubscription messages for all previously received subscriptions. This removes the routes forwarding events across the broken link. Similarly, when a new link is added, the dispatcher sends subscription messages for all of the patterns in its subscription table to the end-point of the new link, thus informing its new neighbor of the events that must be sent

across the new link. These subscriptions and unsubscriptions propagate normally along the tree, updating subscription information.

Figure 5.1 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 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.

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 sub-system 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, 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[1]. In the protocols introduced later, sequential dependencies (depicted by arrows) will be introduced.

5.2 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 describe in Chapter 6.

The Fundamental Problem: *Subscriptions are removed and immediately re-inserted.* The Strawman protocol is the most natural protocol when reconfiguration involves only either an isolated link insertion or removal. However, the most frequent case in a dynamic network is one where a broken link is shortly replaced by a new one. In this case, the Strawman protocol is highly inefficient, as illustrated in Figure 5.2. 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 of the subscriptions in this sub-tree are removed only to be re-added shortly after. This phenomenon generates unnecessary overhead, whose negative

[1] In practice, the overlay sub-system is likely to notify the broken link *before* the new link, due to the distributed search of a new route. However, this is not a requirement of the protocol.

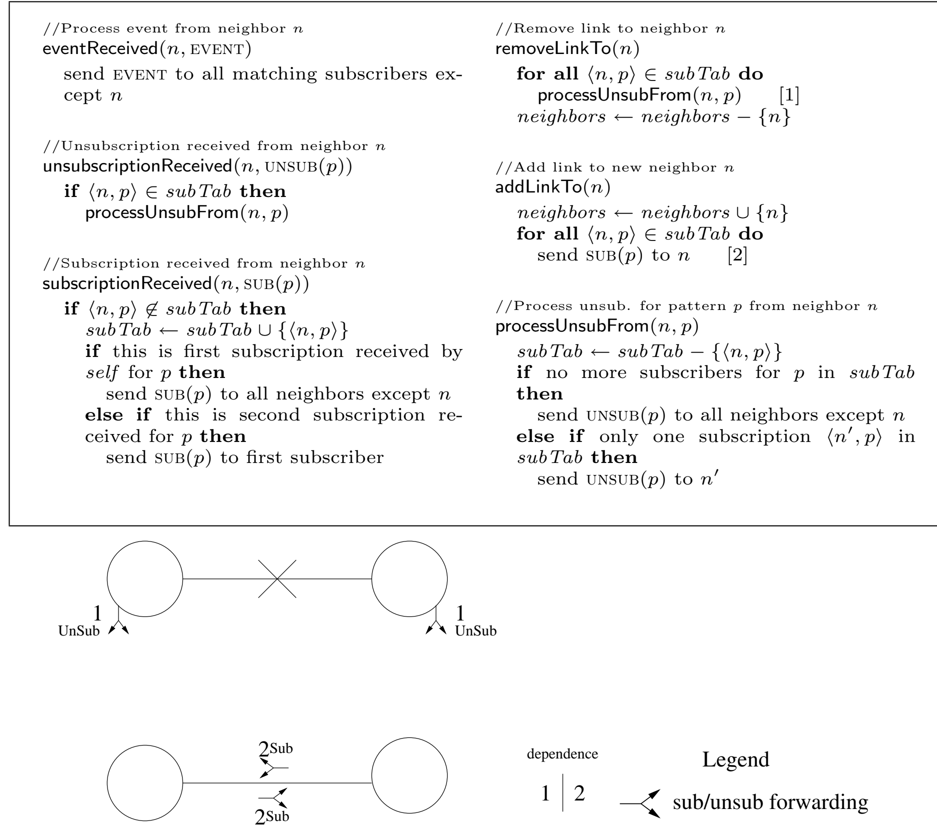


Figure 5.1: Pseudo-code and schematic of the STRAWMAN protocol. The pseudo-code also outlines normal event, subscription and unsubscription processing. Numbers in square brackets indicate the corresponding messages in the schematic below. The schematic assumes the top link is removed and the bottom link is added. To the right of the schematic is a dependence diagram showing implied (but not strict) dependence among steps with numbering. The legend on the far right 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.

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 part of the thesis. Before delving into the details of the protocols 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

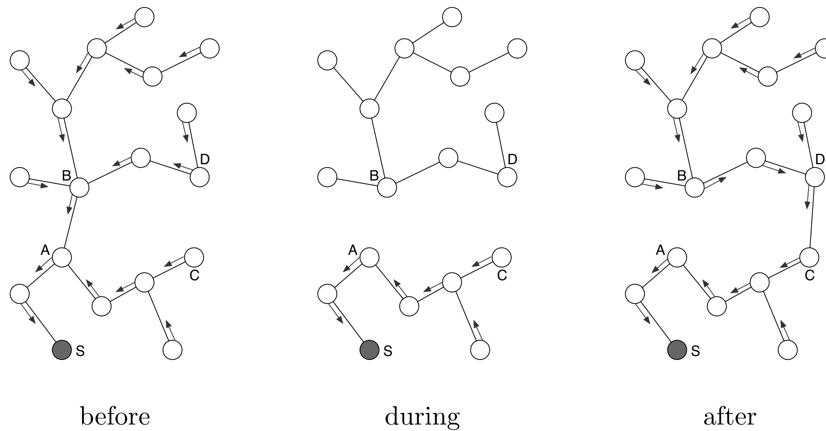


Figure 5.2: A dispatching tree of before, during and after a reconfiguration performed using the STRAWMAN approach.

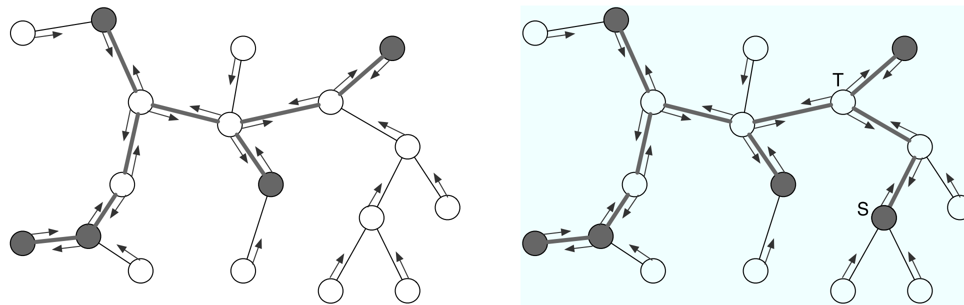


Figure 5.3: Understanding the propagation of subscriptions. In the left figure, the pattern tree for a gray pattern is shown. In the right figure, when a dispatcher S issues a subscription for the gray pattern, its subscription is propagated only up to the closest dispatcher (T) that is part of the pattern tree.

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. The left-hand side of Figure 5.3 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 2.4.1: *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 need to be propagated. Similar rule holds for unsubscriptions: *a unsubscription for a pattern p propagates up to the closest dispatcher that, after having rearranged its subscription table by processing the unsubscription message, remains still part of the pattern tree.*

To understand the former rule 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 the left-hand side of Figure 5.3, 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 the left-hand side of Figure 5.3. 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 we discussed in Section 2.4.1. Instead, if a pattern tree has already been established (even with a single subscriber), as in the right-hand side of Figure 5.3, 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 explain how unsubscriptions are routed.

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 process of reconfiguration that involves the new link. In the STRAWMAN protocol, the end-points of the new link simply send subscriptions for all entries in their subscription table, to ensure that all the events of interest generated in the other sub-tree are properly forwarded. While this is sufficient, however, it is not entirely necessary.

Consider, for example, the situation shown in Figure 5.4 in which only one dispatcher is subscribed to a particular pattern. It may happen that the unsubscription issued by dispatcher B (which will eventually remove the arrows between D and B) propagates slowly, reaching D after the new link opens and after D has exchanged its subscriptions with C. This causes the insertion of extraneous subscriptions (the thick arrows in the middle of Figure 5.4), which will be eventually removed by the slowly propagating unsubscriptions issued by B (right-hand side of Figure 5.4). Therefore, the protocol still correctly restores the routing information, but it does so in an inefficient way. This phenomenon can occur in

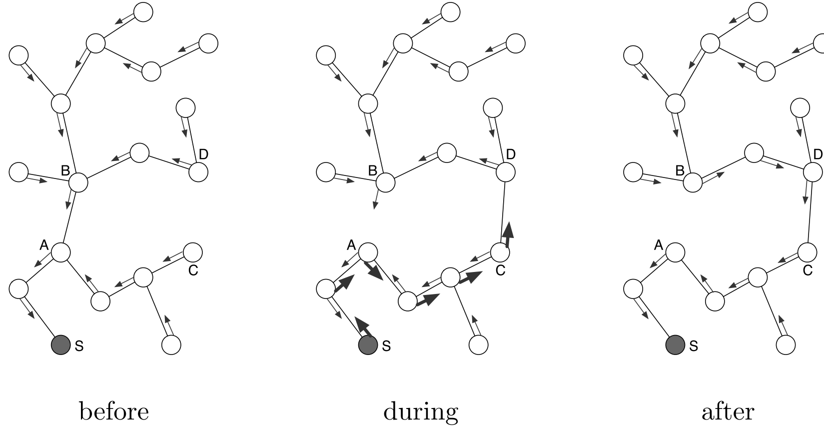


Figure 5.4: Extraneous subscriptions created during reconfiguration when subscriptions precede unsubscriptions.

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 between A and B, 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 generated in Figure 5.4.

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 narrowing the scope of the reconfiguration in terms of dispatchers involved, and therefore helps in designing protocols that limit the impact of reconfiguration. To find which dispatchers are affected 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

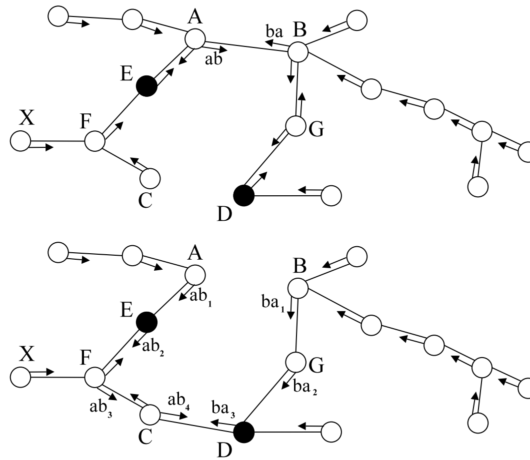


Figure 5.5: A dispatching tree before and after a reconfiguration, showing explicitly the subscriptions that replace the broken link.

the dispatchers connecting it to the second end-point of the removed link, inclusive.

Figure 5.5 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 5.5) 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 reconfiguration.

CHAPTER

SIX

Optimized Protocols

Based on the observations made in Chapter 5, we now describe three optimized reconfiguration protocols: 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 sub-system. 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 sub-system to the routing sub-system 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 the assumptions about the tree maintenance sub-system, overhead reduction capability, and protocol complexity provide the evaluation criteria to decide which protocol to use in a particular environment.

The contribution of this part of the thesis is twofold. We define two new protocols: Reconfiguration Path [36, 40] and Informed Link Activation and we extend previous work [39, 76] by our research group in two different ways. First, we present all the protocols in an integrated and detailed way, with a uniform description in terms of informal pseudo-code. Second we exhaustively compare the protocols against one another through simulation in Chapter 7, and qualitatively in Chapter 8.

6.1 Deferred Unsubscription

As discussed in Chapter 5, 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. The Deferred Unsubscription protocol is based on Observation 1 from Section 5.2: 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 do 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.

6.1.1 Timed Deferred Unsubscription

In the first and simplest variant of the Deferred Unsubscription protocol, shown in Figure 6.1, the delay is provided by a timeout, which is initialized on each end-point dispatcher when a link breaks. The expiration of this timer triggers the propagation of unsubscriptions, therefore we refer to this protocol as Timed Deferred Unsubscription. Ideally, this delay should coincide with the time needed by the underlying tree maintenance sub-system to restore the connectivity of the tree, plus the time required to propagate subscriptions. As in the Strawman protocol, the only assumption required is that the underlying tree maintenance sub-system notifies the end-points of the old and new links.

About Links Sharing a Dispatcher. This protocol enables significant advantages over Strawman. However, from Observation 2 in Section 5.2 we know that the delay of unsubscriptions may lead to unnecessary propagation of subscriptions across the new link. Unfortunately, in this Timed Deferred Unsubscription protocol, we assume that the underlying tree maintenance sub-system does not provide any association between the old and new links, making the optimization outlined in Section 5.2 impossible.

However, in the case where one end-point of the new link coincides with an end-point of the old link (e.g., if in Figure 5.4 dispatchers A and C are the same) 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 management 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 operation manageBrokenLink(n, rId) in Figure 6.1, which is invoked when the link between the current dispatching server and n breaks. When this happens, the filters associated to the neighbor n (i.e., the set of filters $subTab[n]$) are immediately removed

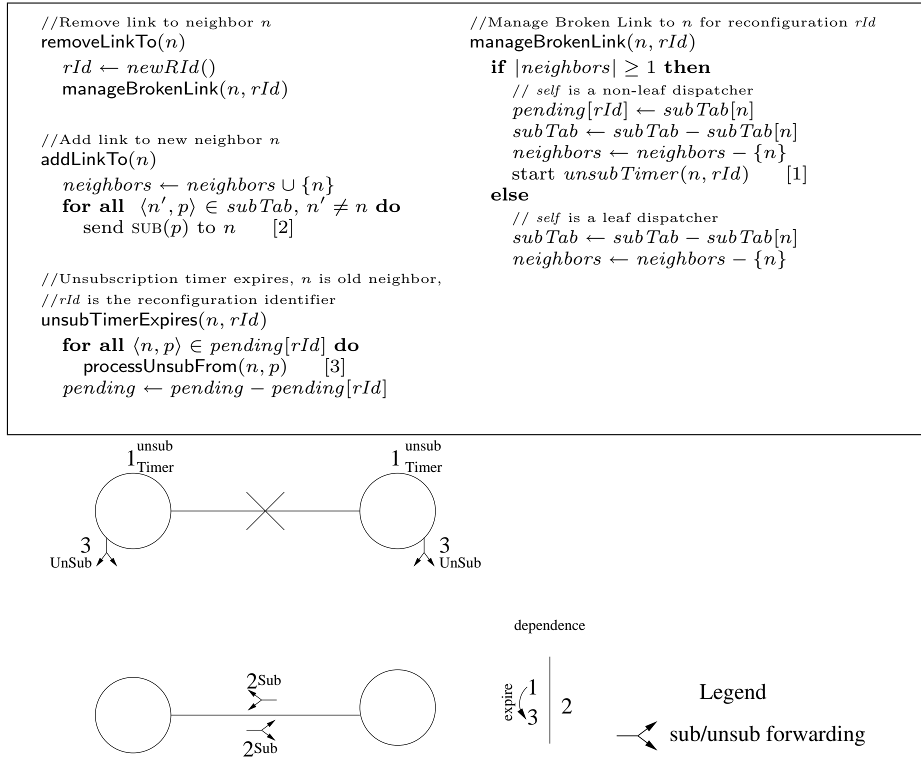


Figure 6.1: TIMED DEFERRED UNSUBSCRIPTION pseudo-code and schematic. Arrows in the dependence diagram show strict dependence. Here and after we use the notation protocol.operation (e.g., strawman.removeLinkTo in the figure) to reuse operations defined in a protocol we already presented.

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 timeout expires, the filters in the *pending* table are propagated, thus enabling the core idea of the protocol, namely deferring the propagation of the unsubscriptions resulting from the reconfiguration.

About the Reconfiguration of 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 which to send messages. As a result it can unsubscribe locally without updating its *pending* table and without setting timeouts.

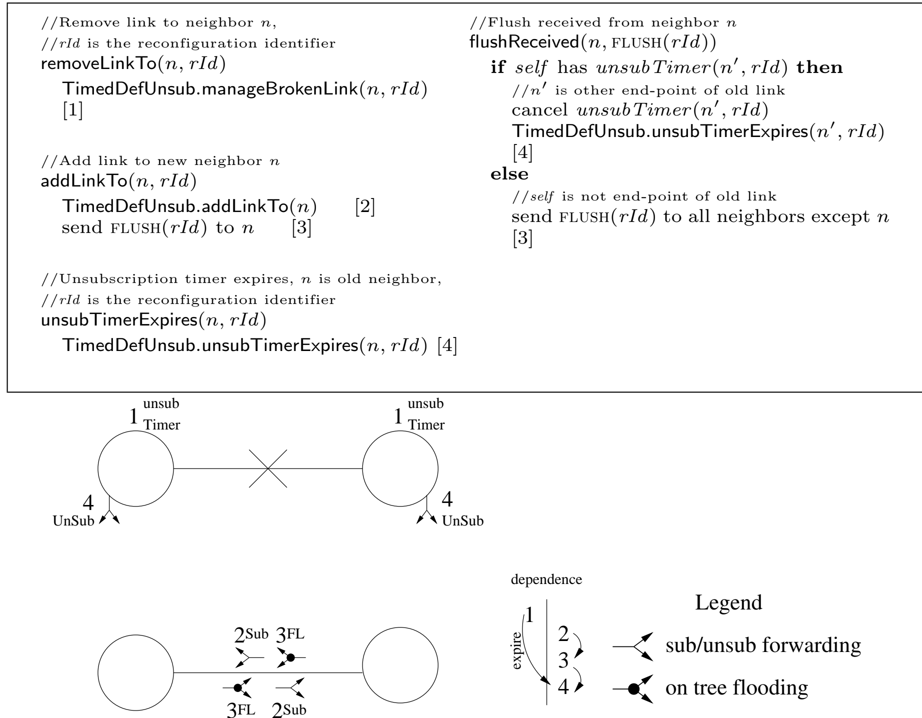


Figure 6.2: NOTIFIED DEFERRED UNSUBSCRIPTION pseudo-code and schematic. In the dependence diagram, a step with more than one incoming arrow indicates it can be triggered by either of the previous steps.

This optimization not only simplifies processing when a leaf dispatcher is involved in a reconfiguration, but it also allows the protocol to avoid the propagation of unnecessary unsubscriptions across the new link when the timeout expires.

6.1.2 Reducing the Dependence on Timers: Notified Deferred Unsubscription

In the TIMED DEFERRED UNSUBSCRIPTION protocol the timeout plays a crucial role. If it 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 an appropriate timer value experimentally for a specific system (as we did in the experiments presented in Chapter 7), we target a dynamic environment where reliance on a statically set timer is inherently approximate. Rather than trying to dynamically adjust the timer, the next protocol introduces a minor modification that complements the timer with a deterministic notification process while still maintaining the benefits of deferring

unsubscriptions. For this modification, which we refer to as NOTIFIED DEFERRED UNSUBSCRIPTION, we assume that the tree reconfiguration sub-system is able to associate the removed link with the inserted link 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 prior to start forwarding subscriptions[1]. These assumptions enable the use of a FLUSH message, which is sent by the end-points of the new link toward the end-points of the old link, just after they finish propagating subscriptions. Since we assumed all links to be FIFO, when the end-points of the old link receive the FLUSH message, they can correctly assume that the subscriptions from the new link have propagated all the way to the old link, and therefore they can start the unsubscription process. This behavior is outlined in Figure 6.2.

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 message serves the same purpose of the timer, namely to start the propagation of the unsubscriptions, it is possible that due to concurrent reconfigurations the FLUSH message does not reach the end-points of the old link. In this case, the unsubscriptions begin propagating when the timer expires.

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 bout the reconfiguration path if it is provided by the tree maintenance sub-system. 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.

6.2 Informed Link Activation

In Section 5.2, 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 even after the reconfiguration. Second, according to Observation 2, subscriptions from the new link may propagate unnecessary 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.

[1]Both assumptions are satisfied by the overlay maintenance protocol presented in Chapter 10

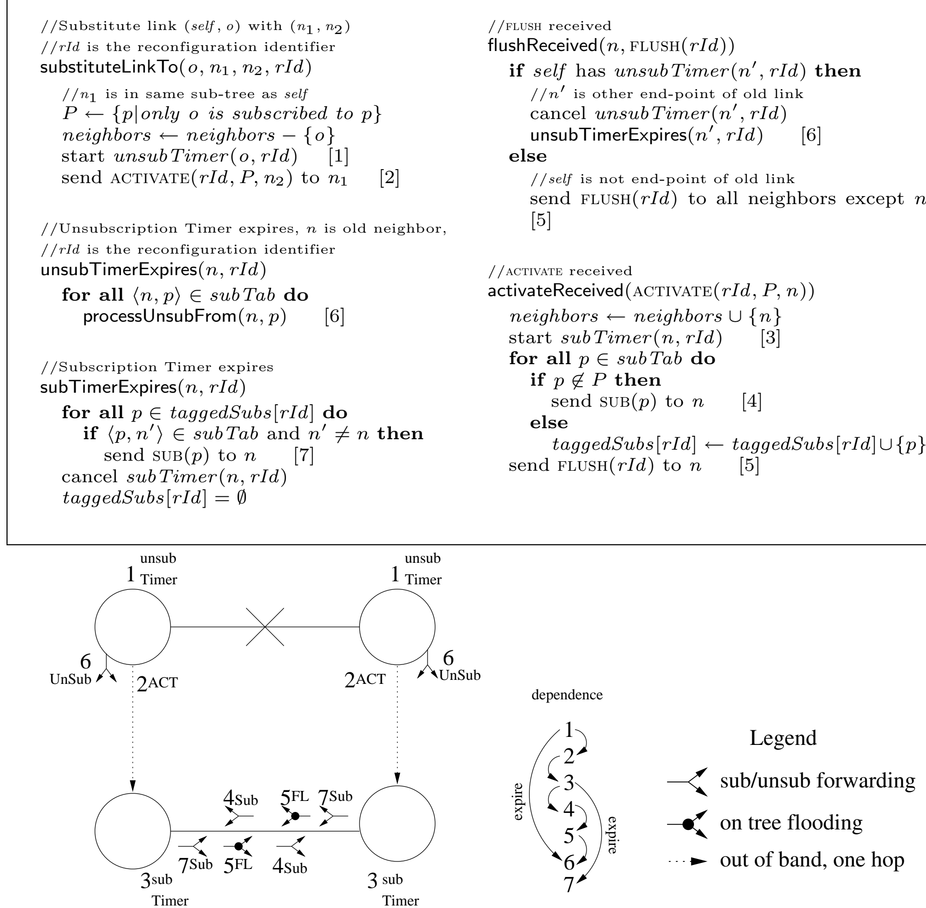
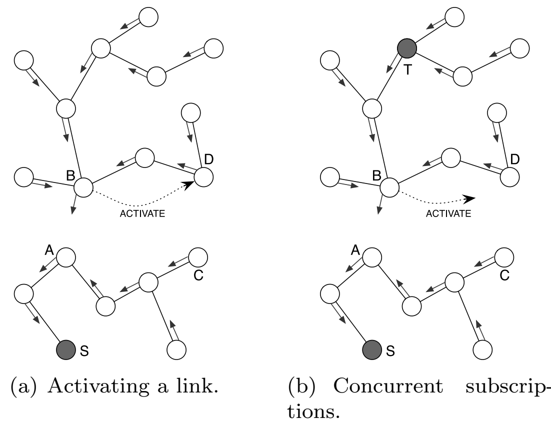


Figure 6.3: INFORMED LINK ACTIVATION pseudo-code and schematic.

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 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 sub-system 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 sub-system.

Figure 6.3 shows the pseudo-code and schematics for the protocol. We assume that the tree maintenance sub-system invokes the operation substituteLinkTo on each of the old link end-points when a reconfiguration occurs. Through this



(a) Activating a link.

(b) Concurrent subscriptions.

Figure 6.4: Two scenarios in the INFORMED LINK ACTIVATION protocol.

operation the tree maintenance sub-system informs the end-points of the old link about the identity of the end-points of the new link. In this operation, each end-point determines the set of patterns P that are used to route events only toward the other sub-tree and propagates it 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 6.4(a) shows a situation similar to the one depicted in Figure 5.4 where, upon breakage of the link (A, B), B sends to D an ACTIVATE message where P contains the pattern for B's subscription towards A. Upon receiving this message, D does not propagate the subscription across the new link, therefore avoiding the generation of the extraneous subscriptions shown in Figure 5.4.

Unsubscriptions can be dealt with as in one of the DEFERRED UNSUBSCRIPTION protocols. In Figure 6.3 (and later in the simulations of Chapter 7) we use the technique described in NOTIFIED DEFERRED UNSUBSCRIPTION, where 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 timeout (set in step 1).

The previously outlined processing is sufficient under stable conditions; that is, when the only subscriptions and unsubscriptions being propagated are those determined by the current reconfiguration. In reality, however, new subscriptions can be generated concurrently with reconfiguration and multiple reconfigurations can occur in parallel. In such cases, some of the subscriptions that have been deemed unnecessary may still need to be propagated. For instance, looking at Figure 6.4(b), dispatcher T may decide to subscribe to the same "gray" pattern as S concurrently to the replacement of link (A, B) with link (C, D). In this case, contrary to what we stated above, 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 these situations, the INFORMED LINK ACTIVATION protocol uses an additional *subscription timer*, which is 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 (i.e., *taggedSubs* in Figure 6.4(b)). To guarantee the right behavior in the presence of multiple reconfigurations, both the timer and the subscriptions not propagated in step 4 are tagged with the reconfiguration identifier *rId*. In our previous example, this leads to the propagation of the subscription issued by T because it has not been removed by the unsubscriptions propagated by B.

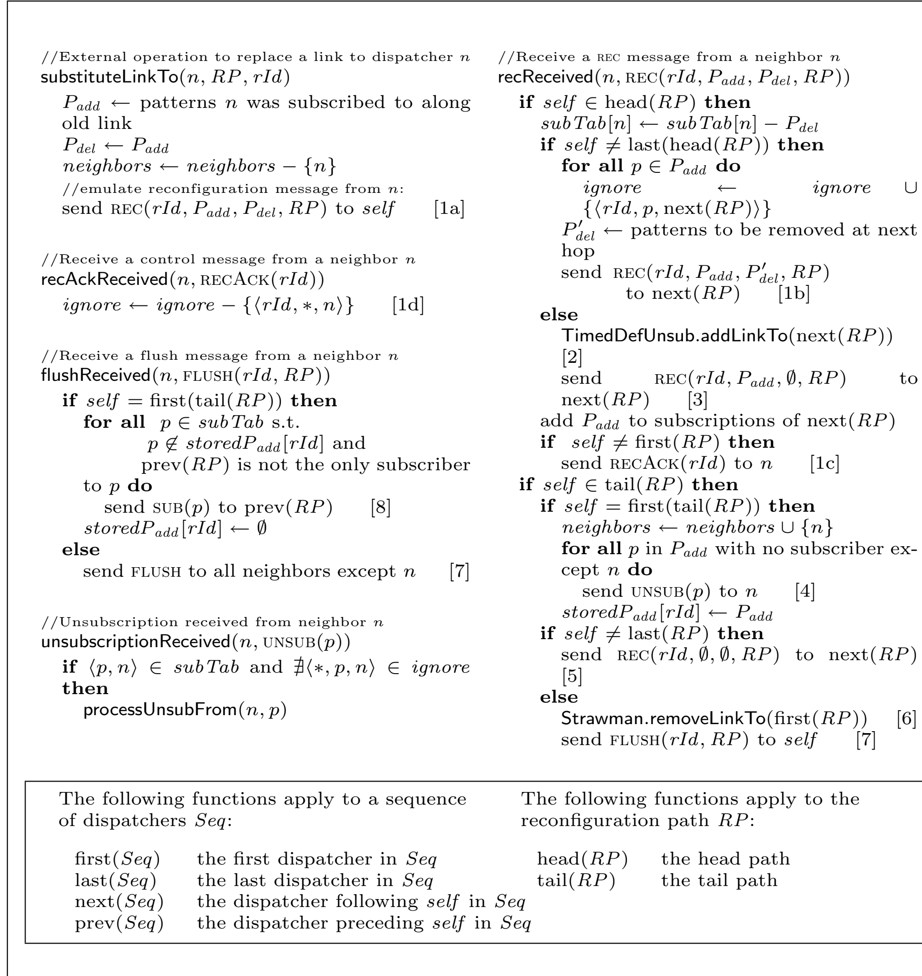
Clearly, the value of the subscription timer must be large enough to allow the unsubscriptions generated at the old link to be propagated to the new link before it expires. If t_{rep} is the sum of the time required by the tree maintenance layer to locate a new route plus the time required for the propagation 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 condition:

$$T_u + t_{prop} < T_s + t_{rep} \qquad (6.1)$$

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 only a small price to pay with respect to the great reduction in overhead achieved by the protocol.

6.3 Reconfiguration Path

Although the previous protocols keep the tree dense of subscriptions to limit the scope of the reconfiguration, it is possible that the subscriptions (and possibly the unsubscriptions) propagate beyond the reconfiguration path, introducing unnecessary overhead. Our next protocol focuses explicitly on the dispatchers on the reconfiguration path, which we assume being computed by the tree maintenance sub-system and communicated to the content-based routing sub-system. 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. On receiving such reconfiguration message, dispatching servers rearrange their subscription table to take into account the changes in the overlay topology. This sequential way of operating reduces the overhead at a minimum, but is also the source of the main weakness of the protocol, i.e., the inability to withstand multiple overlapping reconfigurations. Some of the complexity of the protocol also come from the fact that processing is different on the head and tail paths as well as across the new link, which is coherent with the considerations made at the end of Section 5.2. Moreover, the normal subscription and unsubscription operations must be allowed to continue during reconfiguration, further



```
//External operation to 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:		The following functions apply to the reconfiguration path RP:	
first(Seq)	the first dispatcher in Seq	head(RP)	the head path
last(Seq)	the last dispatcher in Seq	tail(RP)	the tail path
next(Seq)	the dispatcher following *self* in Seq		
prev(Seq)	the dispatcher preceding *self* in Seq		

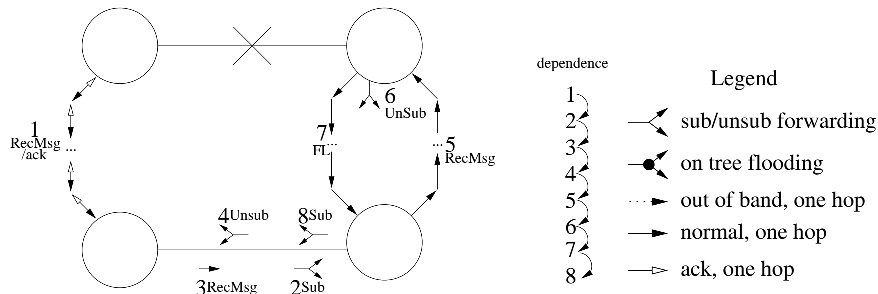


Figure 6.5: Reconfiguration Path pseudo-code and schematic.

complicating the protocol.

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 6.5.

6.3.1 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 is started by the *initiator*, i.e., the first dispatcher on the reconfiguration path, when a call to the operation substituteLinkTo(rId, n, RP) of Figure 6.5 is made by the tree maintenance sub-system[2]. It removes the other end-point of the old link from the set of neighbors and computes the two set of patterns P_{add} and P_{del} that are relevant for the protocol. The former includes the patterns for which subscriptions need to be added along the head path, while the set P_{del} includes the patterns for which subscriptions need to be removed along the head path. With reference to Figure 5.5, 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} are initialized by the initiator with the same patterns: those belonging to subscriptions previously issued by the other end-point of the vanished link (ab in Figure 5.5). Afterwards, the initiator proceeds by simulating the receipt of a REC message (step $1a$ in the pseudo-code).

The message is processed by the recReceived action. For each event pattern in P_{add}, a new entry is inserted in the initiator's subscription table as if it were a subscription coming from the next dispatcher in the reconfiguration path (E in Figure 5.5), thus enabling events to be routed in the direction of the new link. Similarly, all the entries in P_{del} are deleted from the subscription table. In Figure 5.5, these actions cause respectively the insertion of ab_1 and the deletion of ab.

Reconfiguring the Head Path. After reconfiguring its subscription table, the initiator propagates the REC message containing rId, P_{add}, P_{del}, and the list RP 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

[2]Note that differently from the previous protocols the tree maintenance sub-system provides the entire reconfiguration path to the content-based routing sub-system through the parameter RP.

message continues along the head path until it reaches the first end-point of the new link (C in Figure 5.5). This dispatcher behaves differently from the others along the head path because 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 as necessary 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 propagating 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 5.5).

6.3.2 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 approach to reconfiguration.

Avoiding Race Conditions on the Head Path. The first issue arises in the head path. The REC message flows along the head path from the initiator to the first end-point of the new link, while part of its behavior is to add 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.

This is normally acceptable, but it can cause problems when a dispatcher D is already a subscriber for a pattern p and issues an unsubscription for p after the dispatcher D' that precedes D on the head path has inserted p in its subscription table and before the REC message has reached D. The propagation

of the unsubscription from D to D' has the effect of removing the subscription just established by the reconfiguration at D', interrupting the future propagation of events along the reconfiguration path to the second sub-tree.

The solution we adopt requires the sender of the REC message (i.e., D' in the previous example) to remember the subscriptions just 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. Specifically, a dispatcher forwarding a REC message along the head path ignores the unsubscriptions coming from the next dispatcher along the reconfiguration path before the acknowledgement if they correspond to subscriptions just added by the reconfiguration, while it processes the other subscriptions and unsubscriptions normally. In Figure 6.5 this is obtained by using the ignore table (*ignore*) that holds the unsubscriptions that should not be processed, and by adding the acknowledgment message RECACK that has the effect of clearing the *ignore* data structure (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 should check if there are subscriptions generated in 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 are not placed exclusively on the new link and are not contained in $storedP_{add}$ (step 8).

CHAPTER
SEVEN

Evaluation of Routing Reconfiguration

Chapter 6 described in detail the behavior of the optimized protocols considered in this thesis. This chapter complements it by focusing on a numerical evaluation of their performance in several candidate scenarios, using OMNeT++ [95], a popular, open source, discrete event simulation tool. Results highlight the improvements achieved by the two novel solutions Reconfiguration Path and Informed Link Activation, thanks to their ability to restrict changes to the reconfiguration path.

Section 7.1 introduces the simulation environment and the parameters characterizing the scenarios we evaluated. Section 7.2 analyzes the ability of the protocols to restore the correct event routing after reconfigurations, while maintaining reasonable event delivery. Finally, Section 7.3 presents a detailed evaluation of the cost of dealing with reconfiguration for each of the protocols we analyze.

7.1 Simulation Setting

Due to the limited availability of reference scenarios for the application of content-based publish-subscribe middleware, we extended the simulation setting used in [76] and [37]. Clients are not modeled explicitly, as their activity affects only the dispatcher they are attached to and, in addition, in the scenarios we target (e.g., peer-to-peer networks and MANET) the publish-subscribe system is likely to be deployed so that clients and dispatchers coincide, as we pointed out in Chapter 3. The parameters of our simulations and corresponding default values are shown in Table 7.1, 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

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 timeout	$T_u = 0.15$ s
subscription timeout	$T_s = 0.15$ s

Table 7.1: Default simulation parameters. Those in italics remain constant throughout our simulations.

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 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 7.3.3. 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 5.2, 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 7.3.2. Nevertheless, 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., [19]).

In fact, given our event model, the density of receivers for a given event can be computed as

$$\sigma_r = \sigma_s \times p = \sigma_s \left(1 - \left(\frac{\Pi - \pi}{\Pi}\right)^{\mu}\right) \tag{7.1}$$

where p is the probability that a given event matches at least one of a subscriber's patterns. Using the default values in Table 7.1 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 7.3.2 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 part of the thesis 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 is run for 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 7.3.2. The last second is used to allow reconfigurations to complete.

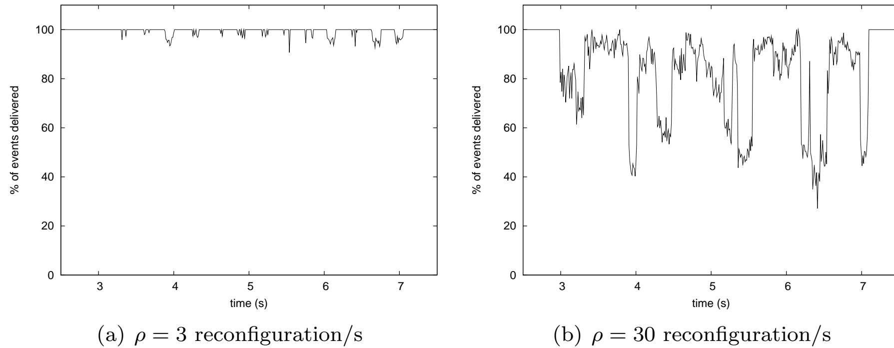
Timers. Some of the considered protocols make use of timers to coordinate their actions. The choice of the best timer values depends on the specific scenario being considered. In the scenario we analyzed, however, we set both the subscription and the unsubscription timers to $0.15s$, as these values yield average performance. An analysis of the impact of timer values is provided in Sections 7.3.2 and 7.3.3.

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.

7.2 Event Delivery

The first property we need to evaluate in routing reconfiguration protocols is their ability to restore correct event delivery 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 exactly 100%. This is exactly the case in Figure 7.1, 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 pres-



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

Figure 7.1: Event delivery for the TIMED DEFERRED UNSUBSCRIPTION protocol; results obtained with the other optimized protocol do not evidence significant differences.

ence 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 7.1 represent 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 7.1(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 7.1(b) leads to a situation where reconfigurations overlap in time and space, therefore negatively affecting event delivery.

Figure 7.1 shows how, independently of the reconfiguration scenario, all of the considered protocols always restore correct routes. Indeed, event delivery goes back to *exactly* 100% after the network topology stabilizes at $t = 7$. Moreover, we verified on our simulation traces that no misrouted events are generated after the routes stabilize.

7.3 Overhead

The simulation results for event delivery indicate that all the protocols we consider correctly restore the routing of events 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 7.3.5 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

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

Table 7.2: Modeling the different costs of publish-subscribe messages and control messages.

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 occurred. This quantity represents the most basic "building block" necessary to assess the behavior of the considered protocols, and in the rest of this section we show how it varies according to a change in the simulation parameters in Table 7.1. 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.

7.3.1 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 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 Chapter 6, 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

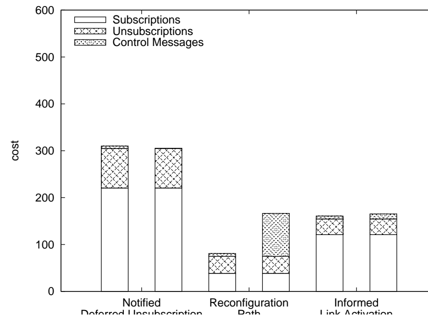


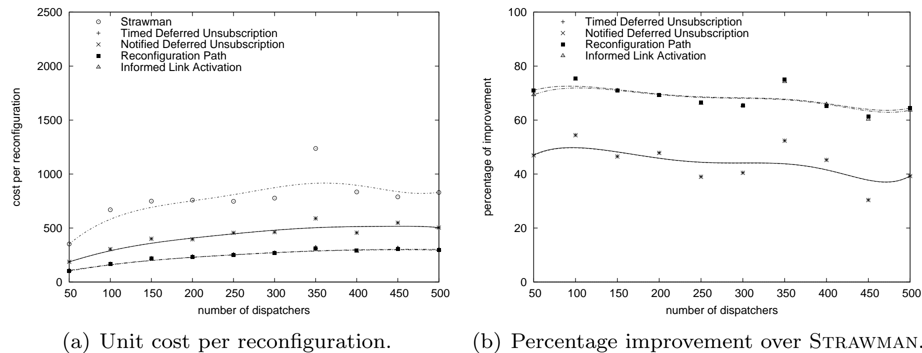
Figure 7.2: Absolute number of messages vs. normalized cost, in a scenario defined by the parameters in Table 7.1 and by $\varepsilon = 0$. For each protocol, the left bar shows the absolute number of messages exchanged, the right bar the normalized cost obtained using the weights in Table 7.2.

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., [93]). 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 7.2. 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 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 7.2, which reports simulation results obtained in the reference scenario defined by Table 7.1. 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 cost of REC and ACTIVATE messages is instead much lower than the normalized one (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.



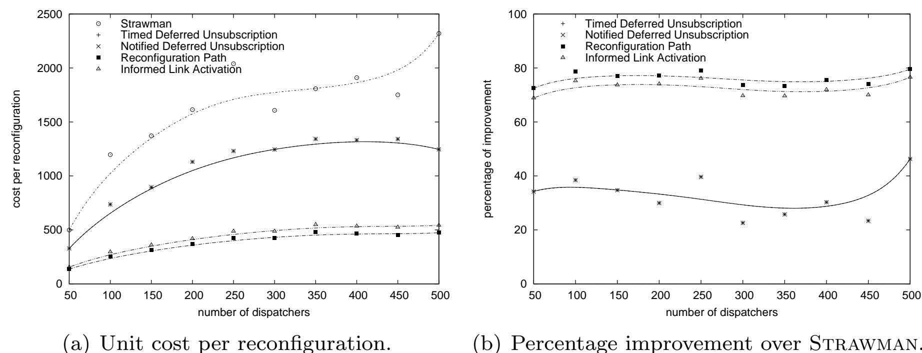
(a) Unit cost per reconfiguration. (b) Percentage improvement over Strawman.

Figure 7.3: Cost of reconfiguration vs. system scale.

7.3.2 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 and control messages. The impact of misrouted events, which is nonetheless negligible in the reference scenario, is analyzed in Section 7.3.3 and following.

System Scale We begin by analyzing the performance of the protocols against the system scale, by ranging the network size N from 50 to 500 dispatchers and keeping the other parameters of Table 7.1 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



(a) Unit cost per reconfiguration. (b) Percentage improvement over Strawman.

Figure 7.4: Cost of reconfiguration vs. system scale, with unbalanced initial tree configurations.

available patterns to $\Pi = 200$ to account for the increased scale. The impact of this latter parameter is analyzed in more detail in Section 7.3.2.

Figure 7.3 shows our simulation results. Figure 7.3(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 indicated by the percentage improvement over STRAWMAN plotted in Figure 7.3(b), all of the considered 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%.

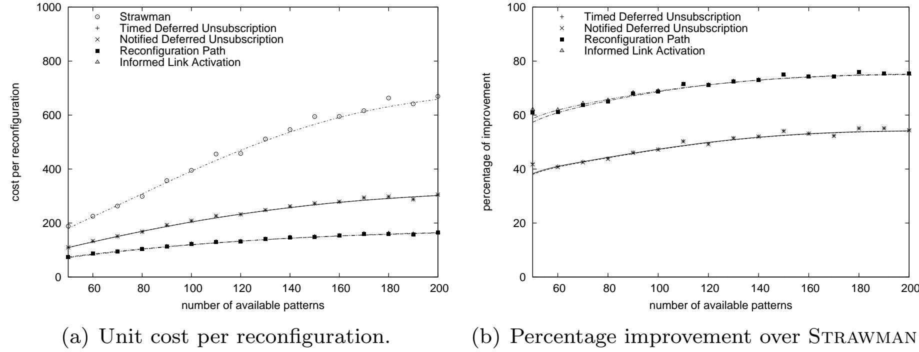
The plots in Figure 7.3 also evidence how the performance of the two variants of DEFERRED UNSUBSCRIPTION is virtually indistinguishable. This is not surprising, as they have the same fundamental behavior, and differ only in the mechanism used to trigger the unsubscriptions previously deferred. Moreover, since the FLUSH messages used by NOTIFIED DEFERRED UNSUBSCRIPTION are small in size and few in number, as discussed in Section 7.3.1, 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 7.3 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 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[1] 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 7.4(a) against Figure 7.3(a). In particular, RECONFIGURATION PATH and INFORMED LINK ACTIVATION 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, RECONFIGURATION PATH and INFORMED LINK ACTIVATION are less affected by the increased diameter of the network because the overhead messages they generate remain confined to the reconfiguration path. Indeed, RECONFIGURATION PATH performs best, since it fully enforces this property. 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

[1] 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.

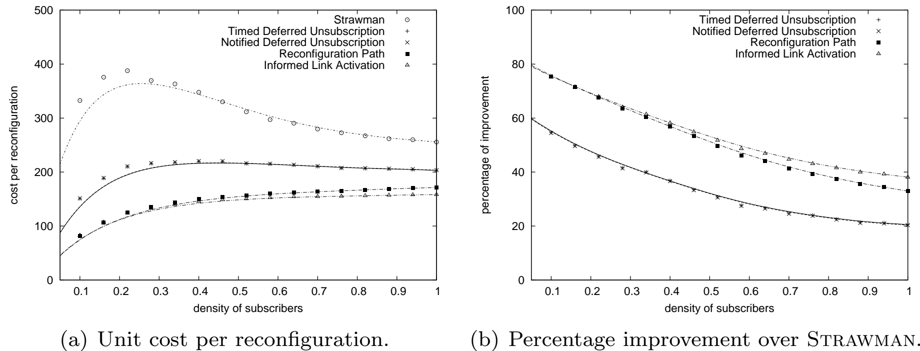
Figure 7.5: Impact of the number Π of available patterns on the reconfiguration overhead.

STRAWMAN are bigger.

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 varying Π in the system affects 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 potentially unique subscriptions. The charts in Figure 7.5, derived for $N = 100$, show that our optimized 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 rather 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, the optimized protocols manage to limit this distance by either deferring unsubscriptions, (i.e., temporarily increasing the density of subscribers) or by explicitly enforcing reconfiguration messages to remain on the reconfiguration path. Figure 7.5(b) confirms this intuition by showing that the improvement of all the 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 7.3.2, to accommodate for the increased scale. Moreover, they also explain why the improvement curve in Figure 7.3(b) exhibits a small decrease when the size of dispatching network increases. This trend is a consequence of our choice of a fixed number of available patterns. When the density of subscribers increases, since the π subscriptions for each subscriber are drawn from the same, fixed set of Π patterns, the tree becomes dense of subscriptions, reducing the total number of hops traveled by subscription and unsubscription messages, and correspondingly reducing 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



(a) Unit cost per reconfiguration. (b) Percentage improvement over STRAWMAN.

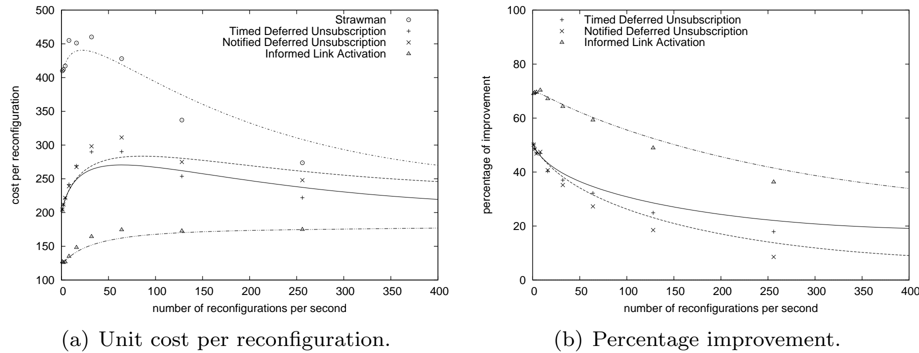
Figure 7.6: Impact of subscriber density σ_s on reconfiguration overhead.

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.

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 5.2. 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 the optimized protocols. Figure 7.6 confirms this intuition by showing the results of simulations in the reference scenario of Table 7.1, changing the density of subscribers in the range $3\% \leq \sigma_s \leq 100\%$, which according to Equation (7.1) yields a number of receivers per event in the range $1.48\% \leq \sigma_r \leq 49.40\%$.

Two things are also worth noting about Figure 7.6. First, even with a tree 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 7.1), as this is one of the aspects differentiating content-based communication from multicast and broadcast communication. Instead, here σ_r is well beyond the values commonly assumed. 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, like those recently made popular by wireless sensor networks [3]. Interestingly, in these applications reducing the communication overhead induced by the monitoring infrastructure is of paramount importance.



(a) Unit cost per reconfiguration.

(b) Percentage improvement.

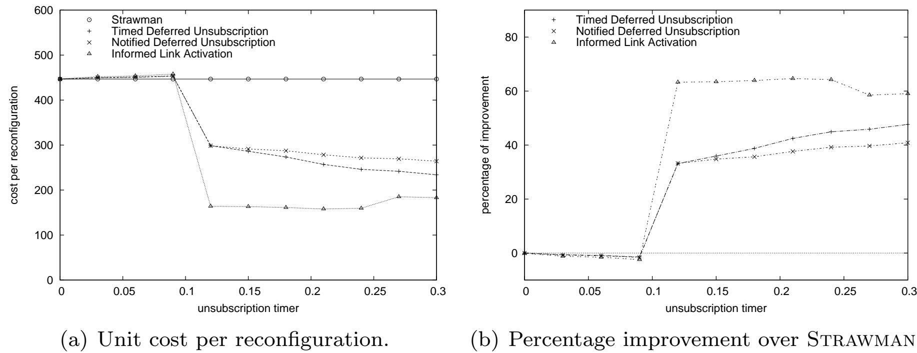
Figure 7.7: Impact of reconfiguration rate on reconfiguration overhead.

Frequency of Reconfiguration Thus far, we 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 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 indeed 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 7.7 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 the protocols in extreme conditions[2].

The charts depict a number of interesting phenomenons, of which the most prominent 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 others operations, i.e., removing subscriptions that are immediately restored by a new reconfiguration, or vice versa. The more a protocol relies on standard propagation 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

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



(a) Unit cost per reconfiguration. (b) Percentage improvement over STRAWMAN.

Figure 7.8: Impact of the unsubscription timer on reconfiguration overhead, at $\rho = 30$ rec/s. The STRAWMAN protocol is shown as a term of comparison.

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 the optimized 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. The difference between the two protocols is that the former triggers unsubscriptions earlier than the latter. As the reconfiguration rate increases, by leaving stale subscriptions in place for longer TIMED DEFERRED UNSUBSCRIPTION benefits from the greater chance that it will be established by another, concurrent reconfiguration, removing the need for an 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 7.7, where the benefits of the optimized protocols are higher.

Timers Some of the considered 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 7.3.3 we instead evaluate their impact on misrouted events when the publish rate is $\varepsilon \neq 0$.

Unsubscription Timer. Figure 7.8 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.

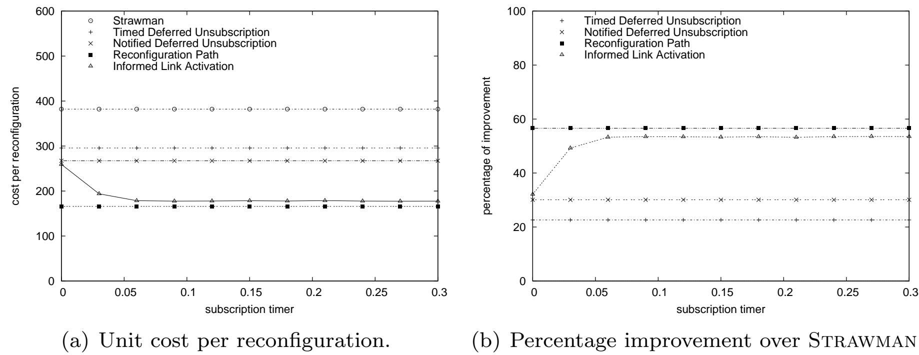
Figure 7.8(a) evidences the presence of a marked discontinuity around $T_u = t_{rep}$, as expected[3]. This discontinuity is *always* present regardless of the value

[3]More precisely, according to equation (6.1), the discontinuity occurs around $T_u = t_{rep} +$

of ρ. If the timer value is too small, the optimized protocols tend to behave in the same way as STRAWMAN because unsubscriptions are triggered too early. Indeed, STRAWMAN is equivalent to either DEFERRED UNSUBSCRIPTION protocols when $T_u = 0$. 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 7.3.2: the removal of a stale route may become unnecessary if another reconfiguration requires establishing the same subscription, and increasing the timer increases the chance that this situation happens. 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. Similar reasoning holds for INFORMED LINK ACTIVATION, as long as the constraint $T_u < T_s + t_{rep}$ we introduced in Section 6.2 holds. If not, i.e., for $T_u > 0.25$ s in our reference scenario, 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 finished propagating, and therefore cause unnecessary overhead. Therefore, we can conclude that for what concerns the cost of reconfiguring subscription tables, as long as $T_u > t_{rep}$ ($t_{rep} < T_u < T_s + t_{rep}$ for INFORMED LINK ACTIVATION) the optimized protocols always improve over STRAWMAN, and they do not bear significant dependency over the value of the unsubscription timer. Nevertheless, the tradeoffs may be different in the presence of event traffic, as we discuss in Section 7.3.3.

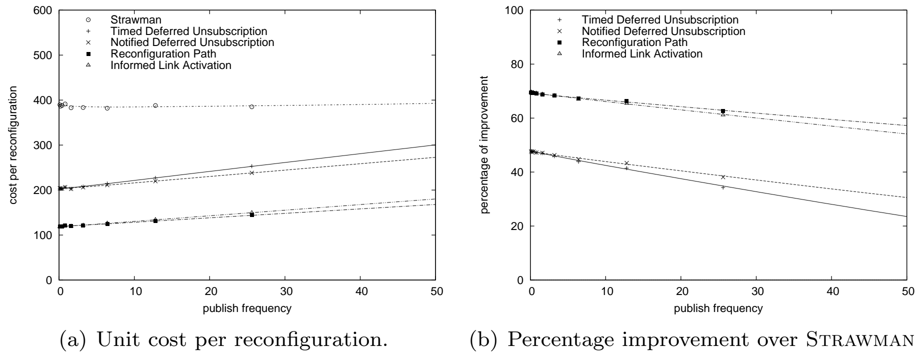
Subscription Timer. Similar considerations hold for the subscription timer T_s employed only by the INFORMED LINK ACTIVATION protocol. This second timer

t_{prop}. However, since the unsubscription propagation time t_{prop} is negligible in our simulations, we do not consider it hereafter.



(a) Unit cost per reconfiguration. (b) Percentage improvement over STRAWMAN.

Figure 7.9: Impact of the subscription timer on reconfiguration overhead. The STRAWMAN, DEFERRED UNSUBSCRIPTION and RECONFIGURATION PATH protocols are shown as terms of comparison.



(a) Unit cost per reconfiguration. (b) Percentage improvement over Strawman.

Figure 7.10: Impact of publish frequency on reconfiguration overhead.

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 6.2) 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 7.9, 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.

7.3.3 Evaluating the Impact of Misrouted Events

We now investigate the performance of the 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.

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 the 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 7.1. 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 MANET applications [57]. The upper bound of more than 50 pub/s is instead al-

most equivalent to a streaming application. Therefore, the high publish frequency in the chart should be regarded mostly as a way to evaluate the protocols in an extreme, and almost unrealistic situation.

Figure 7.10 shows the reconfiguration overhead, while Figure 7.11 reports the absolute number of misrouted events per reconfiguration for each protocol. A different view is provided in Figure 7.12, which exemplifies the relative impact of misrouted events w.r.t. the other components 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 the optimized 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. Nevertheless, 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, it is worth noting, however, that we push the publish frequency to such an unrealistic high event load mostly to stress the performance of the optimized 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 7.13 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 with ε varying in the reference scenario defined by Table 7.1. As it can be seen, in Figure 7.13(a), at $\varepsilon = 1$ pub/s the overhead generated by Strawman is in a ratio of about 1:2 with

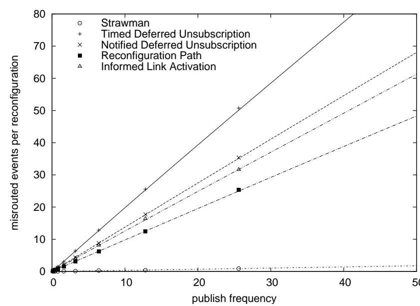


Figure 7.11: Absolute number of misrouted events per reconfiguration.

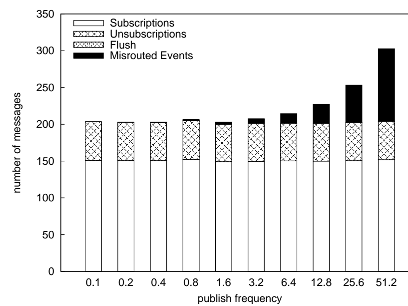
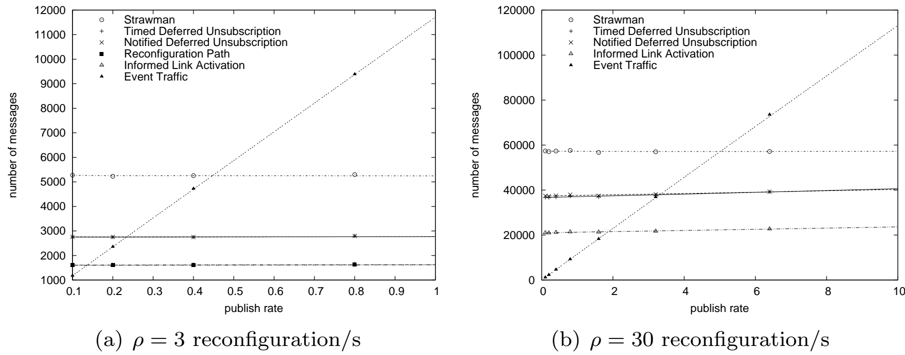


Figure 7.12: Misrouted events vs. the other overhead components in Timed Deferred Unsubscription.



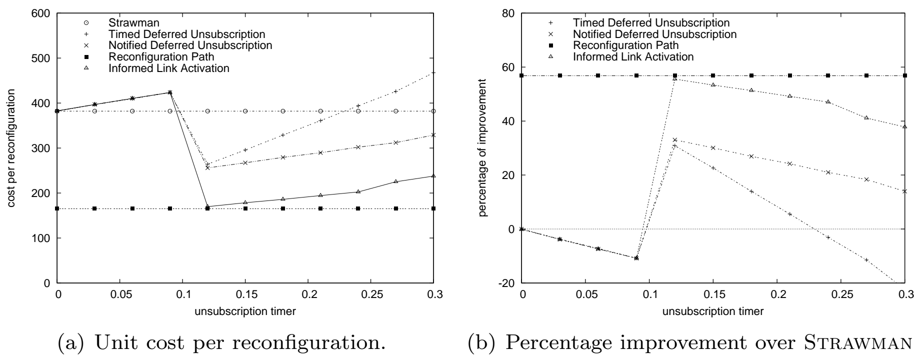
(a) $\rho = 3$ reconfiguration/s (b) $\rho = 30$ reconfiguration/s

Figure 7.13: "Good" event traffic vs. reconfiguration overhead.

the traffic generated by events, while INFORMED LINK ACTIVATION brings this ratio down to 1:10. Smaller values of ε show how the optimized protocols yield even more remarkable improvements. In addition, it is true that the overhead depends not only on ε but also on ρ: as shown in Figure 7.13(b), a higher reconfiguration rate shifts the overhead curves higher, therefore changing significantly the tradeoffs, making our optimizations more relevant.

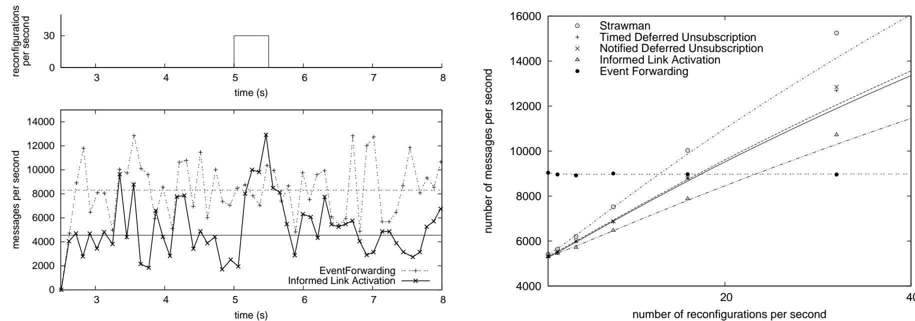
Timers The subscription timer T_s may prevent events from reaching subscribers by delaying the establishment of the corresponding routes, but it bears no effect on misrouted events, and therefore it is not considered here.

On the other hand, the presence of misrouted events may significantly affect the considerations we made in Section 7.3.2 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 our analysis in previous section has shown, however, a very small number of misrouted events is generated at $\varepsilon = 1$ pub/s in



(a) Unit cost per reconfiguration. (b) Percentage improvement over STRAWMAN.

Figure 7.14: Impact of the unsubscription timer on reconfiguration overhead. The STRAWMAN, and RECONFIGURATION PATH protocols are shown as terms of comparison.



(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 overall average number of messages generated outside the burst.

(b) Traffic vs. reconfiguration rate.

Figure 7.15: A comparison against event forwarding. The charts show the total message traffic generated, with a density of subscribers $\sigma_s = 0.8$.

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 the optimized protocols by amplifying the effects of timers, we use a very high publish frequency $\varepsilon = 50$ pub/s.

Figure 7.14 confirms the above reasoning. As in Figure 7.8, there is a discontinuity around $T_u = t_{rep}$. Differently from Figure 7.8, 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, it 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 (FLUSH messages) fails.

7.3.4 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 2.4.1. 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 2.4.1. 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.

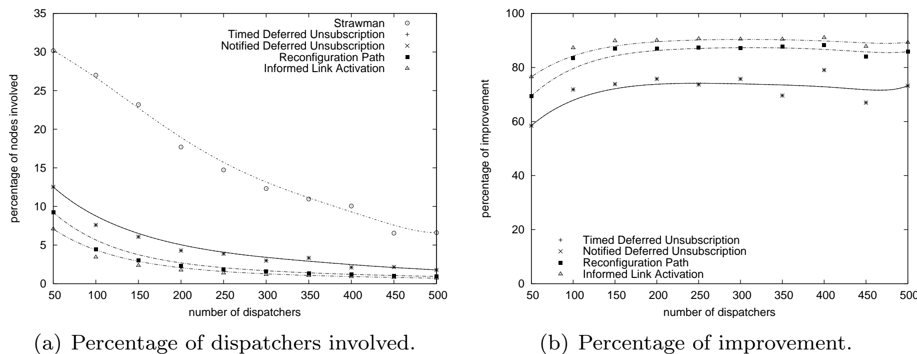
Figure 7.15(a) crisply illustrates the tradeoffs at stake by showing the total traffic generated by event forwarding and subscription forwarding, the latter ex-

tended with our INFORMED LINK 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 7.1. The horizontal lines represent the average total number of messages 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. As an example, Figure 7.15(b) plots the overall message traffic against ρ for $\varepsilon = 1$ pub/s, i.e., in the same conditions of Figure 7.15(a). If the value of ρ is a reasonable one (unlike those we used in the reconfiguration burst above or in Section 7.3.2) then subscription forwarding is always preferable over event forwarding, with our protocols providing significant additional enhancements.

7.3.5 Computational Overhead: Dispatchers Involved in a Reconfiguration

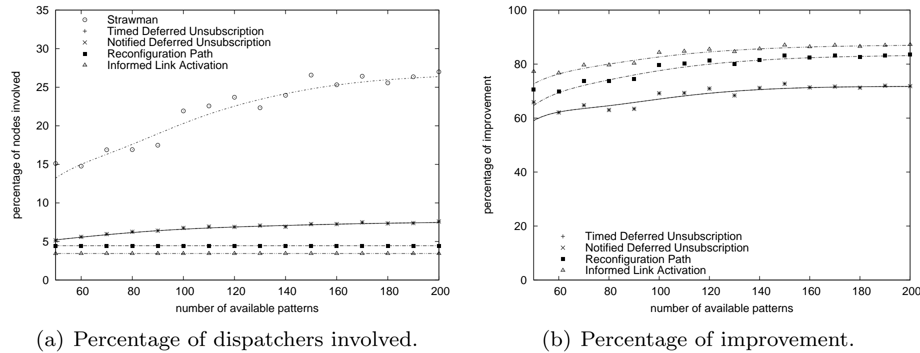
Thus far, we considered only the overall communication overhead induced by reconfiguration. Nevertheless, another way to look at the burden reconfiguration places on the system is to examine the computational overhead induced on the dispatchers. Our simulations do not capture this directly, and in any 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,



(a) Percentage of dispatchers involved.

(b) Percentage of improvement.

Figure 7.16: Dispatchers involved in a reconfiguration vs. system scale.

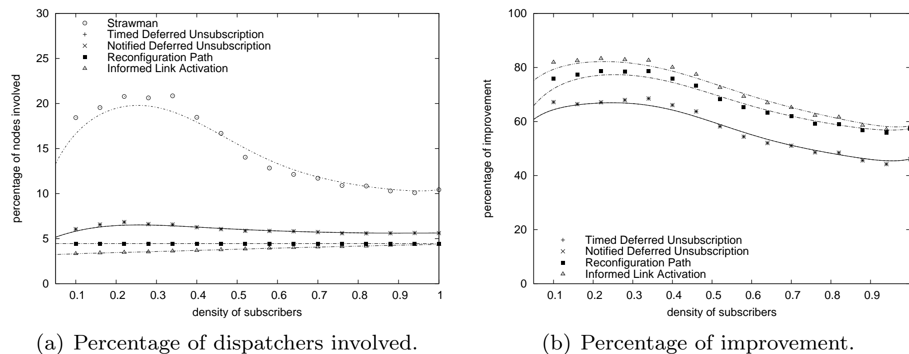


(a) Percentage of dispatchers involved. (b) Percentage of improvement.

Figure 7.17: Dispatchers involved in a reconfiguration vs. number of available event patterns.

we consider a dispatcher as *involved* in a reconfiguration if it performs processing related to the reconstruction of routes disrupted by it. This clearly includes 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. Instead, we do not consider 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 Chapter 6, confirming that our protocols are able to significantly limit the portion of the system involved in a reconfiguration. For



(a) Percentage of dispatchers involved. (b) Percentage of improvement.

Figure 7.18: Dispatchers involved in a reconfiguration vs. subscriber density.

instance, the fraction of dispatchers involved is shown against system scale in Figure 7.16 using the same parameters of Figure 7.3. All of the optimized 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, similarly to what we discussed in Section 7.3.2, an unbalanced initial configuration amplifies the differences among the approaches.

Another interesting perspective is provided by Figure 7.17 and 7.18, which plot the dispatchers involved against the density of subscribers and number of available patterns, respectively, with the same setting of Figure 7.5 and 7.6. 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.

CHAPTER
EIGHT

Discussion and Related Work

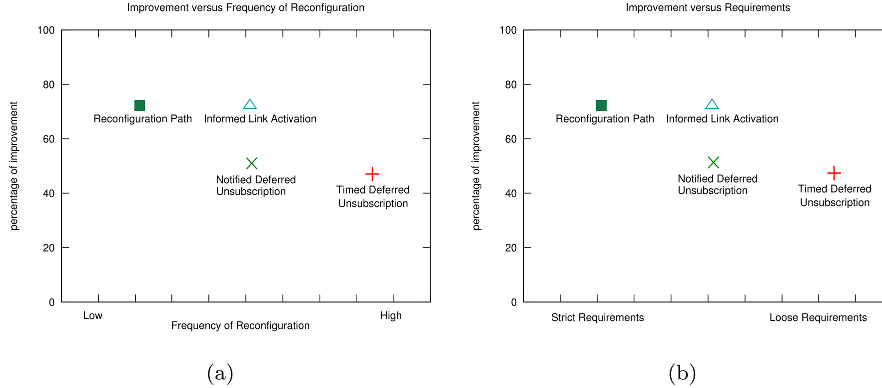
The previous chapter characterized the performance of routing reconfiguration protocols along several dimensions, therefore providing a useful and immediate way to compare quantitatively the various solutions. Nevertheless, each protocol bears strengths and weaknesses, determined by the assumptions it 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 Chapter 7.

In this chapter, we address this problem and analyze the differences between the optimized protocols from a qualitative standpoint. In addition, we consider existing alternative approaches to address content-based routing in dynamic network scenarios and discuss their strengths and weaknesses with respect to our own. Finally we close our analysis of routing with some concluding remarks.

8.1 Applicability Versus Performance

In this section we complement our quantitative simulation results with qualitative observations about the applicability and complexity of the solutions we presented. Together, these considerations enable one to choose the most appropriate protocol for a given deployment scenario.

Figure 8.1 and 8.2 show the main differences among the protocols we analyzed in this part of the thesis. Figure 8.1 plots the improvement achieved over Strawman against the supported application scenarios and against the requirements on the overlay, while Figure 8.2 shows the differences in a tabular form. The first column in the table refers to the ability of the protocols to tolerate multiple, concurrent reconfigurations, 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



(a) (b)

Figure 8.1: Performance of the protocols with respect to applicability.

about the operation of the underlying overlay network sub-system. 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. The Notified Deferred Unsubscription and Informed Link Activation assume that the overlay network sub-system 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 information is provided by most overlay management protocols, such as the one presented in Chapter 10, and the one we will use in Chapter 16 for deploying our middleware in mobile ad hoc networks [67]. Finally, the Reconfiguration Path protocol requires knowledge of the whole sequence of nodes belonging to the reconfiguration path. This requires either dedicated protocols (e.g., straightforward variants of our aforementioned protocols) or more stringent assumptions on the deployment scenario (e.g., manual reconfiguration performed by a system administrator).

		Arbitrary Changes?	Knowledge of Topology	Additional Messages	Complexity
Strawman		Yes	A link appears or disappears	None	Low
Deferred Unsubscription	Timed	Yes	A link appears or disappears	None	Low
	Notified	Yes	A link replaces a given vanished link	flush	Low/Medium
Informed Link Activation		Yes	A link replaces a given vanished link	activate, flush	Medium
Reconfiguration Path		No	Nodes on the reconfiguration path	rec, recAck	High

Figure 8.2: Comparing the applicability and complexity of reconfiguration protocols.

Together, these two columns are a qualitative indicator of the applicability of the protocols to different scenarios. This is clear with respect to the ability to tolerate arbitrary reconfigurations: as we already mentioned in Section 6.3, this is a precondition for applicability in highly dynamic environments, e.g., mobile computing and peer-to-peer networks, which are therefore precluded to the RECONFIGURATION PATH protocol. Nevertheless, the amount of required knowledge about the changes in the underlying overlay network is also an indirect measure of applicability, in that it poses deployment constraints (i.e., the availability of an overlay management module with given characteristics) that may not be satisfied by a given system architecture.

The third column shows how many 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 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 Chapter 6.

These qualitative considerations, together with the quantitative evaluation we presented in Chapter 7, enable us to distill some conclusions. To begin with, TIMED DEFERRED UNSUBSCRIPTION is the least efficient of the protocols we analyzed. 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 particularly in the presence of high event traffic. 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 sub-system. 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 perfor-

mance and applicability.

8.2 Related Work

The considerations we made in Section 8.1 show that our reconfiguration protocols cover a broad range of reconfiguration scenarios, ranging from human-controlled reconfigurations to very dynamic network environments.

In this respect, the work presented in these chapters fills a fundamental gap in content-based publish-subscribe research. Existing middleware addresses reconfiguration only to a limited extent. To the best of our knowledge, no previous work has described and analyzed a variety of reconfiguration techniques such as those presented in this part of the thesis and evaluated in Chapter 7.

Nonetheless, some existing work has partially addressed the problem of content-based routing in dynamic network scenarios either with techniques similar to ours or using completely different approaches such as those based on epidemic information dissemination. Moreover our work on content-based routing is related to other research area such as multicast and more in general group communication, even though the need to route messages based on their content makes content-based routing a more challenging problem. In this section we discuss these related approaches and highlight their differences with respect to the solutions presented in this part of the thesis.

8.2.1 Distributed publish-subscribe systems

After the first experiences, which mostly focused on local area networks and adopted a centralized dispatcher, recent years have seen the development of a number of content-based publish-subscribe systems that exploit a distributed dispatcher. Among the most widely known are Siena [18], Gryphon [11], Hermes [77, 78], Xnet [25], REBECA [47], Jedi [35], Joram [10], NaradaBrokering [73], Le Subscribe [45], READY [52], Elvin [86], and TIBCO Rendezvous [92].

Most of these systems do not provide any explicit mechanism to reconfigure the dispatching infrastructure in reaction to changes in the underlying network. A first exception to this situation is provided by Siena [18] and the system described in [98], which briefly mention the use of the Strawman solution to allow sub-trees of brokers to be merged or trees to be split. Both these works neither provide details about the design of this facility, nor validate it through simulation.

A more efficient solution to the problem of rearranging the dispatching network is provided by Hermes [77, 78], which provides a slightly limited form of content-based routing, termed "type and attribute based" routing [43]. Hermes uses techniques developed in 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. The self-organization and stabilization features of the Hermes peer-to-peer substrate handle dynamic addition, removal, and failure of brokers. However, the overhead involved in the Hermes approach to reconfiguration has not been analyzed.

While in our research we adopted a reactive approach, based on the idea of rearranging content-based routing each time a change occurs at the networking layer that could impact the routing layer, the solutions described in [68] and [19] adopt a proactive approach, based on the idea of periodically refreshing subscriptions. More specifically, in [68] an explicit lease time is associated to subscriptions. Clients are required to refresh their subscriptions once per leasing period, while dispatchers discard routing entries that have not been renewed in time. Similarly, in [19] subscriptions are periodically renewed and re-propagated, while an appropriate protocol is periodically run to remove stale subscriptions that may result from changes in the networking topology. Both approaches provide resilience to changes in the underlying network at the cost of requiring continuous refresh of subscriptions, a cost that can be very high. Also, the time required for the system to reach a stable situation after a perturbation depends on the frequency of refresh, which cannot be shortened below a certain value to keep the overhead under control. A reactive approach like the one we adopted is not affected by these problems and should provide better performance in all those scenarios where reconfigurations are not too frequent.

In addition to the systems above, which explicitly consider the same reconfiguration problem we focus on, other systems focus on fault-tolerance and reliability. In particular, Xnet [25] provides several mechanisms, including the use of redundant routes, to reduce the impact of dispatcher crashes. Similarly, Joram 4.2 [10] and the extension to Gryphon described in [100] allow a set of dispatchers to operate as a single redundant cluster to deal with link failures and dispatcher crashes. Both approaches provide a limited form of reconfigurability with respect to that offered by the protocols studied here.

Finally, Jedi [35], the extended version of Siena presented in [17], Elvin [91], REBECA [69, 46], and the work described in [79] support a different scenario where clients are enabled to roam freely by detaching from one dispatcher and attaching to another. However, none of these works address the reconfiguration of the dispatching network itself.

8.2.2 Publish-subscribe on MANETs

Recently, the problem of rearranging the dispatching infrastructure of a publish-subscribe system has been addressed by different research groups in the context of MANETs.

Most solutions in this area, like [101, 65, 97, 34, 7] do not even try to build and maintain an overlay network for content-based routing, so they cannot be directly compared with the protocols described in this paper.

Other solutions, like [58, 87], focus on replication of dispatchers to overcome the challenges stemming from nodes' mobility. As in the case of the distributed publish-subscribe systems described above this provides a fairly limited form of reconfigurability with respect to that offered by the protocols studied here.

Finally, JEcho [27] takes advantage of an underlying unicast protocol to build a tree shaped overlay for routing on a MANET without caring about topology changes due to mobility. To overcome the limitations of this layered approach, which could easily result in an overlay network whose topology does not reflect

that of the physical network, JEcho dispatchers periodically run a link state protocol to build a global view of the physical network, rearranging the overlay accordingly. Apparently, JEcho does not provide explicit mechanisms to manage node partitions and node failures, which could require a rearrangement of routing tables, like in the cases we focus on.

8.2.3 Other research areas

In addition to publish-subscribe middleware, our work is related with IP multicast routing protocols and group communication middleware.

The purpose of IP multicast 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 one 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, while in the systems we are interested in 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 [71], where several alternatives are compared.

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, and in addition to guarantee some degree of ordering among events and an atomic behavior [29]. Under this umbrella fall systems providing reliable multicast facilities [62, 70] as well as more complex systems such as Isis [14] and Horus [94], which provide several communication primitives to coordinate a set of distributed components. In contrast to the goals provided by group communication systems, the main purpose of content-based publish-subscribe systems is to distribute events to all the interested parties based on their content and to do so in a scalable and efficient way. This difference has a strong impact on the underlying protocols and mechanisms adopted. For example, the implementation of most group communication systems distributes information by using either point-to-point connections from each source to each of the group members or IP multicast. Both of these approaches have drawbacks when applied to content-based publish-subscribe middleware. The former because it does not scale well, and the latter because, as already mentioned, it is very difficult to use IP multicast strategies to efficiently route events based on content.

8.3 Concluding Remarks

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 part of the thesis, 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, of which two constitute contributions of this thesis, that extend the common subscription forwarding strategy with the ability to tolerate topology changes in a number of application scenarios. These 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 75% without hampering the ability to deliver event notifications to interested parties.

As mentioned in Chapter 3, the protocols we presented are designed to be integrated with an overlay management layer. In the remaining parts of this thesis we first present our solutions for maintaining the overlay and then address the combination of the two layers in a single middleware framework. The results we obtain integrate the detailed treatment we provided in this chapters, evidencing additional tradeoffs and further opportunities for optimization.

Part III

Overlay Layer

CHAPTER
NINE

Overlay Requirements and Goals

In this part of the thesis we address the lower layer of our middleware architecture: the overlay manager. Overlay topologies are prominent in large scale distributed computing as they abstract the low-level network into an application-level entity, under the control of the application or middleware. This allows developers to build their systems over a higher-level abstraction ignoring details like node discovery or connection management.

The management of an overlay topology above the network stack does not allow network layers to intervene and address changes resulting for example from node or link failures. Rather, the overlay manager must be ready to address these events directly, maintaining a topology that satisfies the requirements of the upper middleware layers. The focus of this part of the thesis is therefore the maintenance of a tree-based overlay for content-based publish-subscribe middleware. Specifically, we present two novel approaches to maintain the network of brokers connected in a large scale peer-to-peer setting.

From the previous discussion on routing in publish-subscribe middleware we can derive two properties of the mechanisms built on top of the overlay layer. First, on a publish-subscribe tree, events are not sent individually from the source to every destination (e.g., events are not multiply-unicast). Instead, they are sent one time by the source and follow the links of the tree to reach interested subscribers. In the worst case, events traverse every link in the tree one time. In general, though, destination nodes are clustered on the tree so the middleware can save transmission overhead by sending messages once on the links toward the cluster and copying and forking them only when a tree branch is reached with destinations along both branches. This ability to save communication cost is similar to that of routing on a multicast tree, except that in our case routing is further complicated by the need to filter messages according to their content. Second, all event routing information is maintained locally in all the strategies discussed to this point. Each node only knows to which of its neighbors it should

forward events. Beyond this, it does not know which nodes are subscribed for what patterns, or even which nodes are in the tree.

As our goal is the maintenance of the tree, we must pay attention to these properties and the demands they place on the maintenance process. For this reason, we define a set of requirements which must be kept in mind during the design of an overlay protocol for publish-subscribe.

Connectedness and Acyclicity The first requirement is clearly the maintenance of a connected and acyclic topology in spite of arbitrary changes in the underlying physical network. Both the protocols we present exploit mechanisms to prevent the formation of cycles, thereby avoiding the use of expensive distributed cycle detection algorithms. To address connectedness, on the other hand, each protocol exploits a specific approach. The one in Chapter 10 maintains a set of node caches which allow it to replace failed neighbors in all practical cases. The one in Chapter 11, exploits the redundancy characterizing Distributed Hash Tables to replace failed interconnections.

Clearly, maintaining connectivity in the presence of failures is only feasible if a sufficiently large number of nodes remain alive. When this does not happen, partitions may occur; nevertheless the protocols must be able to return to a connected topology as soon as possible.

Management of Node Degree The second requirement descends from the properties of event routing described above. Specifically the cost to a dispatcher for forwarding an event notification is proportional to the number of its neighboring dispatchers. Thus to minimize this cost, the overlay topology should not have nodes with too large a degree. If this happens, nodes with too many neighbors risk to be burdened by the computational cost associated to forwarding messages, while those with a few neighbors such as leaf nodes incur very low cost. The maximum degree of each node should therefore be limited by taking into account its processing power.

Localized Recovery The third requirement aims to minimize the cost of reconfigurations at the routing layer. The routing information maintained at each node needs to be modified whenever the underlying overlay topology changes. As a result, changes determined by the disconnection of a node should be localized to a small region in the vicinity of the disconnected node. This restriction reduces the effort for reconfiguring routing information in publish-subscribe systems, thereby making the middleware significantly more efficient.

Quick Recovery The need to continue message propagation even when topology changes occur requires that the overlay layer be able to reconnect the overlay quickly after a disconnection or a failure. A short reconfiguration period limits the number of event notifications that may be lost due to reconfiguration and thus greatly simplifies the job of the event-recovery layer.

Following these requirements we designed two overlay management protocols that offer good performance not only in combination with the content-based publish-subscribe routing mechanism, but also as a basis for subject-based publish-subscribe, application-level multicast, and other middleware paradigms. In the following we describe these protocols: Chapter 10 presents LSTree, a protocol explicitly designed for the efficient maintenance of large-scale overlay trees, while Chapter 11 presents DHTree, a DHT-based solutions that can reproduce a reference topology on the network using a DHT as a mapping mechanism. Finally, Chapter 12 discusses the differences between the two protocols and their relation with existing work.

CHAPTER
TEN

Large-Scale Tree Maintenance

In this chapter we propose LSTree, a novel strategy to address the requirements described in Chapter 9 by keeping a network of nodes connected in environments characterized by a high frequency of node failures. The key aspect of the protocol, presented in [49], is the ability to provide middleware with an overlay topology that guarantees both efficiency and reliability. This is achieved with a quick and communication efficient reconfiguration process that allows nodes to respond to the disconnection of their neighbors. The protocol is particularly suited for application in the context of publish-subscribe middleware due to its ability to manage node degree and to keep reconfigurations localized to small regions around failed nodes. This is confirmed by simulations results that also show its increased performance with respect to other solutions presented in the literature.

The chapter is organized as follows: Section 10.1 provides a high-level description of our tree maintenance protocol. Section 10.2 describes the mechanisms used by nodes to ensure that the overlay remains an acyclic structure with the desired properties. Section 10.3 details the strategies used by nodes to reconnect the tree after a failure. Section 10.4 provides an in-depth analysis of the proposed approach through simulation, and finally Section 10.5 concludes the chapter with some remarks about possible future developments.

10.1 High Level Maintenance Protocol

With the LSTree overlay maintenance protocol, we aim to organize network nodes in a rooted tree structure while promptly reacting to the connection and disconnection of nodes. The root node is selected during the protocol operation and all nodes, including the root, are allowed to join and leave the overlay at any time without any explicit departure announcements. The choice of a rooted structure does not limit the applicability of LSTree. Rather it makes it suitable for

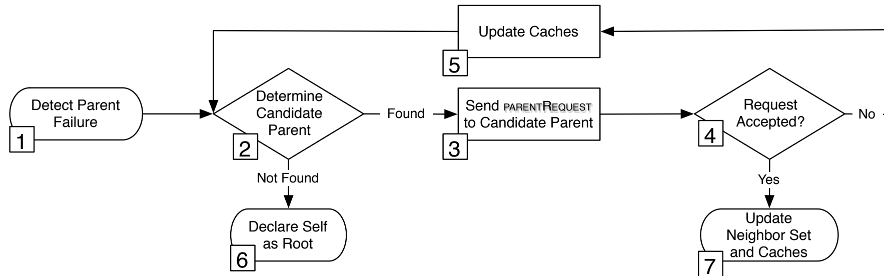


Figure 10.1: High level protocol to locate a new parent after the current parent fails.

middleware requiring either a rooted or an unrooted tree topology.

The protocol operates in a fully distributed manner. Each node records the identities of its one-hop neighbors (parent and children) in the overlay in a *neighbor set*. Application-level beacons are used to monitor the members of this set to detect disconnections. When a node with one parent and n children leaves the overlay, the protocol repairs the tree through the insertion of n new links. The children of the disconnected node take an active role and each of them locates a new parent. Similarly, when a new node joins the overlay, it searches for a parent and establishes a single new link.

The search for a new parent constitutes the core of the maintenance protocol and is outlined in Figure 10.1. A node detecting the failure of its parent determines the identity of a node that might serve as a replacement (step 2) and sends it a ParentRequest message (step 3). The candidate parent is responsible for deciding whether it can safely accept the request without creating cycles and without violating other requirements of the overlay. If the request is accepted (step 5), the two nodes update their neighbor sets and the process terminates successfully. Otherwise, the candidate parent sends a refusal message specifying the reason for the refusal to the requesting node, which reacts by selecting another candidate and restarting the process.

These steps are iterated either until a request is accepted or until the requesting node is unable to locate any new candidates. In the latter case, the node assumes that no suitable parent exists and it elects itself the new root (step 6). This can happen in two cases. The first and most common is the disconnection of a root node. The children of the failed root cooperate to repair the tree with one declaring itself as the new root. The second case arises in scenarios with failures of large sets of nodes, leading to temporary partitions of the overlay. The LSTree protocol merges such partitions by having root nodes periodically search for nodes in a different tree. More details about the declaration of new root nodes and the merging of trees are provided in Section 10.3.3.

The two key aspects of the protocol just outlined are the strategies used by requesting nodes to locate new candidates in step 2 and the ability of candidate parents to accept and refuse connection requests in step 4. In the following

sections we consider each of these aspects and describe how they determine the characteristics of the overlay and the performance of the reconfiguration process. First, in Section 10.2, we describe the mechanisms used by candidate parents to determine whether connection requests should be accepted. Then, in Section 10.3, we discuss several strategies to locate a new candidate parent and describe how they are exploited by the LSTree protocol.

10.2 Determining Parent Viability

Candidate parents play an important role in step 2 of the protocol because their decision to accept or reject ultimately affects the properties of the overlay. In the following we describe these properties distinguishing between a hard requirement, which must always be satisfied, and a soft requirement, which, although desirable, may be overridden to allow reconnection.

10.2.1 Hard Requirement: single acyclic tree

The primary purpose of a tree maintenance protocol is to organize network nodes in a single, acyclic tree. To ensure cycle freedom, we define a partial ordering relationship $n \rightarrow p$ (read *can-connect-to*) that expresses the correctness of n to be a child of p, based only on information known to n and p. This property is always guaranteed between all parent/child pairs, and thus transitively for the entire tree. In fact, a candidate parent p will reject a ParentRequest from another node n in step 4 of the protocol if $n \rightarrow p$ is not true.

A viable definition of $\rightarrow$ is based on an integer value maintained at each node representing its current distance (hop count) from the root. A node without a parent can connect to any node with a hop count *less than* its own. Such a connection is based only on information known at the two nodes and cannot introduce cycles.

This approach is used, for example, in the mobile ad hoc wireless network protocol MAODV to prevent the creation of cycles in multicast trees [84]. Nevertheless, it suffers from two main drawbacks that make it unsuitable for the large-scale scenarios we target. First the use of integer hop counts from the root prevents nodes from connecting to candidates that are at the same distance from the root even though this would not necessarily create a cycle. Second, it forces nodes that change their distance from the root to propagate their new distance to their descendants, thus incurring significant global communication overhead on each topology change.

To address these concerns, our cycle-prevention mechanism exploits real-valued depth instead of integer hop count. Moreover, our LSTree protocol integrates the prevention of cycles, with the ability to merge previously partitioned trees with limited overhead. For this, in addition to *depth* each node is assigned a *color*, both of which are described in the following.

Real-valued depth. We define the *depth* of a node as a real number that represents a relative distance to the root. The depth of a child must always be greater than that of its parent, as shown in Figure 10.2. Introducing real numbers yields

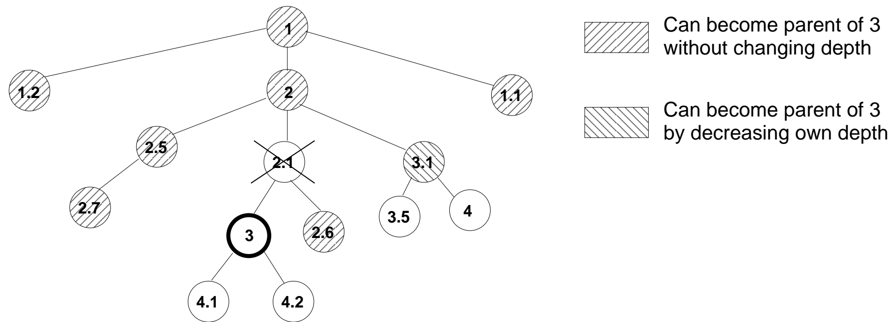


Figure 10.2: Sample depth levels (unique to dual as identifiers). Caches are shown for node 3, whose parent has just failed. All nodes in this example have the same color.

a flexibility over integer hop counts. Specifically, the relationship between the depth of a parent and that of its children is not restricted to *plus-one*; but it must still retain the $\rightarrow$ relationship.

With real-valued depths we address the two requirements outlined above: (i) allowing connections that are not possible with the use of hop-counts, and (ii) preventing the frequent propagation of new depth values to large portions of the tree. To illustrate this, consider a node that attempts to repair the failure of its parent by connecting to one of its former siblings. With integer hop count this connection would be forbidden even if it does not create cycles because the requesting node is at the same distance from the root as the candidate parent. Instead, with real-number depth, the candidate parent may already have suitable depth, e.g., node 2.6 in Figure 10.2. Moreover, if its depth is too large, it may be able to decrease it, making it less than that of the requesting node but still greater than that of its own parent. The figure illustrates the same option applied to a non-sibling node. In this case, node 3.1 can decrease its value to 2.9, making itself a valid parent for 3. In general, decreasing depth allows candidate parents to accept connection requests that would have otherwise been rejected.

To guarantee the $\rightarrow$ relationship remains consistent with the partial ordering defined by the tree structure, we allow nodes to decrease their depth values, but never to increase them.[1] This has two important consequences. First nodes need not coordinate with their existing child nodes when decreasing their depth. Second, nodes need not have perfect information about the latest depth of their parent, but instead can safely compare their own depth against a previously cached parent depth. If their depth is greater than the cached value, then it is also greater than the parent's current depth. This guarantees that the protocol behaves correctly even when two neighboring nodes modify their depths concurrently.

A node learns the depth of its parent when it first connects to it. In addition parent nodes piggyback their depth information on all control messages sent to their children.

Tree Color. Depth provides a simple mechanism to determine whether a candidate parent can accept a node from the same tree as one of its children. *Color* allows nodes to identify that they are members of separate trees (i.e. trees with different

[1] Depth may only be increased when changing color as we discuss later.

roots), triggering tree merging by making the root of one tree the child of a node in the other tree.

Our goals for merging trees are to prevent cycles and to avoid requiring coordination among all nodes in the overlay network. We achieve this by exploiting a relationship between colors, denoted $\rightarrow$. This defines a total order between different trees, which we use to define that the root of one tree can connect to any of the nodes in the other tree, but not vice versa. Establishing this connection involves only the root of one tree and a single node in the other tree.

The color of a tree is established by its root, and we guarantee that if the root changes, the color also changes. Color values are propagated from parent to child, and must remain consistent with respect to the $\rightarrow$ relation. Satisfying this constraint drives two aspects of the LSTree protocol. First candidate parents evaluate whether to accept connection requests by using a combination of color and depth. Second, because color changes propagate from the root node to the rest of the tree and $\rightarrow$ must always hold between a parent and a child, a node with color C_1 can only change its color to a new value C_2 such that $C_1 \rightarrow C_2$.

To allow these parent to child updates, we express a color as a sequence of node identifiers and define the $\rightarrow$ relationship as follows: $C_1 \rightarrow C_2$ if and only if either (i) C_2 is a longer sequence than C_1; or (ii) C_1 and C_2 are sequences of the same length and $C_1 > C_2$ according to standard string comparison. We also define a set of rules for the creation and propagation of color values. First, the initial color of a detached node is an empty sequence, allowing it to become a child of any node with a non-empty color. Second, root nodes generate new colors by appending their identifiers to their current color. This always happens when a node becomes the new root of a tree, but can also be done spontaneously as we discuss later. Third, each time a node updates it color, it must propagate the new color to all descendants, allowing the entire tree to eventually acquire the same color. Finally, a node connecting to a new parent with a different color accepts the color of that parent and propagates it to its descendants.

Exploiting Color and Depth. The LSTree protocol exploits both color and depth to prevent the creation of cycles when adding a link between two nodes. To accomplish this, we extend the $\rightarrow$ to include both components: given two pairs, (C_A, D_A) and (C_B, D_B), $(C_A, D_A) \rightarrow (C_B, D_B)$ holds if and only if $(C_A \rightarrow C_B) \vee (C_A = C_B \wedge D_A > D_B)$. Further, we enforce the following invariant:

$$\textit{For every parent}(C_p, D_p)\textit{and child node}(C_c, D_c), (C_c, D_c) \rightarrow (C_p, D_p).$$

This invariant is considered when a candidate parent evaluates a ParentRequest. If the invariant is satisfied, the request is accepted. If not, but the colors are the same and the candidate can *safely* decrease its depth to satisfy the invariant, it does so and accepts the request. Otherwise the request is rejected.

Decreasing depth is safe except when the candidate is itself searching for a new parent. In this case, the candidate replies with a special refusal message with a *busy* flag set, informing the requesting node that it may later resubmit its request.

One negative consequence of decreasing depth values is that node depths tend to converge to that of the root. To limit this phenomenon, we periodically allow

trees to redistribute their depth values. This is done with the root selecting and propagating a new color C' such that $C \rightarrow C'$. Upon receipt, each node updates its color and sets its depth to that of its parent plus a random number. Such an update is possible because the $\rightarrow$ definition involving both color and depth gives higher priority to colors. Therefore by properly selecting C', the propagation maintains the $\rightarrow$ relation. It should be noted that this update involves all nodes in the overlay; however redistributing depth values is a rare event.

10.2.2 Soft Requirements on Node Degree

In addition to strictly enforcing cycle freedom, candidate parents also try to manage their node degree, preventing too few or to many neighbors. This is important, for example, when applying the LSTree overlay manager in the context of our publish-subscribe middleware. In this case, the costs to deliver a message include the total number of hops it travels (a value that increases if nodes have low degree) and the per-node cost proportional to the number of neighbors to which the message can be forwarded (a value that increases if node have high degree).

One step to manage this requirement is the introduction of a maxDegree limit, expressing the desired maximum number of neighbors for a network node. This value can be defined independently for each node according to its computational capacity. If a candidate parent has already reached its limit, it can refuse a connection request, forcing the would-be child to choose a new candidate parent.

A second limit, instead, aims to avoid the opposite situation in which a large number of nodes have few neighbors. Such a situation can result because low-degree leaf nodes represent a large percentage of the nodes in the overlay and are therefore likely to be chosen as candidate parents. Therefore, we introduce a minimum degree limit, minDegree, to actively discourage connection to low-degree nodes. In our experiments we set the value of this limit to 2, meaning that a candidate parent can accept a ParentRequest if its degree is greater than or equal to two; that is, it is not a leaf node.

Even though reasonable node degree is desirable, it is more important to maintain a single, acyclic tree. Therefore, unlike cycle-freedom, minDegree and maxDegree are considered *soft* requirements, meaning that users of the LSTree overlay protocol can choose whether or not to enforce them and to override them when they would conflict with other more important requirements.

10.3 Identifying Candidate Parents

The key component of the LSTree protocol that has not yet been defined is the identification of candidate parents. References to candidates are stored in a set of caches maintained at each node. This section describes these caches and a corresponding set of strategies that exploit their contents to repair the tree. We also show how these strategies are combined to form protocol instances with different properties, and we conclude with the role of the strategies in recovery from root failures and merging separate trees.

Cache	Update Policy
Regional	Updated by parent when regional nodes change
Downstream	Updated with data from candidate parent refusing due to MAXDEGREE
Upstream	Updated with data from candidate parent refusing due to MINDEGREE
BreakMaxDegree	Updated with reference to candidate parent refusing due to MAXDEGREE
BreakMinDegree	Updated with reference to candidate parent refusing due to MINDEGREE
Global	Proactively maintained by each node

Table 10.1: Caches and their update policies.

10.3.1 Repair Strategies

The policies for updating the caches are related to the functionality of the tree repair strategies. Table 10.1 outlines the caches and their update policies and the remainder of this section provides additional detail including the desirable tree properties each strategy is designed to achieve.

Although the information held in the caches ages, the decision to accept or reject a connection request lies with the candidate parent. Therefore, because the PARENTREQUEST contains the color and depth information of the requesting node, out of date information in the cache of the requester cannot affect protocol correctness. Further, no bandwidth must be spent to maintain consistency.

Repairing Failures Locally: Regional Strategy. The first repair strategy we consider aims to limit the impact of overlay maintenance on the middleware and applications deployed at higher layers. This involves localizing the topology changes resulting from the disconnection of a node to a small region of the overlay. In the case of content-based publish-subscribe, for example, topology changes force the middleware to rebuild part of its routing information. If changes are localized, they affect only a small fraction of this information and thus reduce the overall cost of the reconfiguration process.

Therefore, the Regional strategy proposes new candidate parents in the immediate vicinity of the failed node. The strategy extracts candidate parents from two caches: the *Ancestor Chain* and the *Sibling Set*, collectively referred to as the *Regional* cache. The Ancestor Chain contains n nodes starting with the parent and continuing toward the root while the Sibling Set contains all nodes that are also children of the parent node. Whenever this information changes as a result of reconfiguration, the parent sends an update to its children.

The regional strategy faces two options for selecting a candidate parent: a sibling or an ancestor. If all nodes use only the Ancestor Chain, the resulting topology tends to converge towards a *star* topology in which a few nodes have a high node degree. Conversely, if all nodes (except one) choose to connect to siblings, the resulting topology tends to resemble a line. To avoid these extremes, we opt for a compromise that selects either the star or line option with a specified probability, having some nodes connect to a sibling and others to an ancestor.

Locating Nodes with a Low Degree: Downstream Strategy. As previously discussed, preventing nodes from reaching high degree is often desirable. However, both the use of Ancestor Chain nodes and the need to find nodes with low depth tend to select parents close to the root, which therefore tend to increase in node degree

and eventually refuse connection requests due to maxDegree. This motivated the development of a strategy to search in the opposite direction, namely farther from the root, to nodes more likely to have smaller degree. It is worth noting that we verified this supposition through simulation.

The Downstream strategy achieves this by exploiting a Downstream cache populated with the downstream neighbors of candidate parents that have refused a new link due to high node degree. The number of downstream neighbors sent with each refusal and the total size of the cache are tunable. Clearly, this strategy is only applicable if the maxDegree limit is enabled, as it is the maxDegree refusal message that contains data to populate the Downstream cache.

Locating Non-Leaf Nodes: Upstream Strategy. Our LSTree protocol complements the maximum degree limit with a corresponding minimum degree limit. If a candidate parent refuses a connection request because this minimum limit (i.e. it is a leaf node), the requesting node is likely to find a better candidate by searching higher.

The Upstream recovery strategy provides a way to locate such a new candidate by collecting in an Upstream cache the parents of the nodes refusing a ParentRequest due to the minimum degree limit. Analogously to the previous strategy, the use of the Upstream strategy is only possible possible if the minDegree limit is enabled.

Softening Degree Constraints: BreakMaxDegree and BreakMinDegree Strategies.. Even though maintaining a reasonable degree is desirable, there may be cases in which it is less important than other properties such as the locality of reconfiguration. In such cases, a local connection that overrides the minimum or maximum limits may be preferred to a non-local connection that satisfies the limits.

The *BreakMaxDegree* and *BreakMinDegree* strategies make this possible and force a candidate parent to accept a ParentRequest even if the limits are broken. Nodes that refuse a request because of degree are added to the BreakMaxDegree or BreakMinDegree caches. When the corresponding strategies are activated, requests are sent to candidates with flags set to require acceptance regardless of current degree. Clearly, candidate parents may still refuse requests if their color-depth pairs are incompatible.

Recovering from Catastrophic Failures: Global Strategy. The repair strategies described thus far require first contacting at least one node in the regional cache to bootstrap the process and begin filling the other caches. However, there may be cases in which all the candidates in a node's regional cache become unreachable. To handle such cases, we exploit a Global cache that keeps references to a selection of nodes in the overlay that are not necessarily close.

Clearly, a trade-off exists between the size of the Global cache, the possibility that the cache contains stale references to nodes that are no longer in the overlay, and the overhead for updating the cache. Ultimately the choice is middleware and application specific, and the population of the Global cache may be done with information from these higher layers. Within this thesis, we strike a balance among the requirements with a simple Global cache update mechanism in which nodes exchange messages containing a random subset set of references extracted from their caches. Each update message is sent to a subset of nodes extracted from

the same caches. A node receiving an update inserts the references in its Global caches and optionally re-propagates the update. In addition, nodes periodically ping the nodes in their Global caches to verify if they are reachable, purging those who do not respond.

It should be noted that, as with the other caches, the Global cache does not need to contain up-to-date information about candidate parents. As a result the frequency of updates and ping probes can remain low as discussed in Section 10.4.1, thus limiting bandwidth consumption.

10.3.2 Combining Recovery Techniques

Clearly, each of the strategies we described targets a particular desirable characteristic for the tree, however the behavior of the protocol as a whole depends on the ways in which the strategies are combined. Specifically, we define a protocol as a sequence of strategies. When selecting a candidate parent, the first strategy in the sequence whose cache is non-empty provides a candidate. A ParentRequest is sent and if this node accepts, the protocol completes, however if it rejects, the caches are updated and the selection repeats.

In theory, the strategies can be combined in arbitrary ways, however some combinations make more sense. For example, in most applications, nodes over the maxDegree limit are less desirable than low-degree nodes, hence BreakMaxDegree should be applied only after BreakMinDegree has failed. Also, because locality is important, Global should only be applied after all local repair attempts have failed, that is after Regional, Upstream, and Downstream.

A protocol instance is completely defined as a sequence of strategies by and whether or not the degree constraints are enabled. In the instances studied in Section 10.4, the minimum degree limit is enabled only if the protocol instance contains the BreakMinDegree strategy. Moreover, all of our instances include the Regional, the Global, and BreakMaxDegree strategies.

10.3.3 Declaring New Roots and Merging Trees

The recovery strategies of the LSTree protocol provide nodes with references to candidate parents. However, the failure of a root node, a physical partition in the network, or a very high failure rate may cause all strategies to fail, leaving a node without a parent. When this happens, the node declares itself as a new root, defines a new color and propagates it to its descendants as described in Section 10.2.1.

This same mechanism allows LSTree to recover from root failures. In this case, the children of the failed root connect to each other using their Sibling Sets; however, one of them will fail. Therefore, it terminates its recovery process by declaring itself as a new root.

The declaration of a new root node may occasionally result in the creation of multiple trees, however LSTree also takes care of merging these separate trees with the same mechanism it uses to locate new parents. Each root node periodically attempts to locate a new parent by sending ParentRequest messages to the nodes in its Global cache.

A node N receiving a ParentRequest from a root R reacts as it would with any other request. Let C_N and C_R be their respective colors. If $C_R = C_N$, then the nodes are or have recently been (if a new color is still propagating) in the same subtree. Thus, N will certainly have a higher depth than R and will simply reject the request. On the other hand, if the two nodes are in different trees, two cases may arise. If $C_R \to C_N$, then N accepts the request and merges the two trees by becoming the new parent of R. Otherwise, if $C_N \to C_R$, N rejects the request and informs its own root, which then continues the merging process by contacting a node in R's tree.

10.4 Simulation Study

This section complements the description of the LSTree protocol with a detailed simulation study using OmNet++ [95]. Our analysis verified that LSTree maintains a single acyclic tree in the face of arbitrary disconnections. Additionally, it compares different combinations of recovery strategies with respect to their ability to manage node degree, localize changes, and provide a quick reconfiguration process.

10.4.1 Simulation Setting

We evaluated the LSTree protocol in two different scenarios: the first to test the approach under normal operations, *Gnutella*, and the second to evaluate the system under extreme stress, *Catastrophic*. The former is generated from real trace data indicating the online time of nodes in a Gnutella network [85][2]. The latter consists of an artificial network of 10,000 nodes in which, once the topology has stabilized, 2500 nodes simultaneously fail. To reduce the effects of random parameters we ran 10 simulations in each scenario and averaged the results. As we discuss in the following, LSTree correctly maintains an overlay with the desired properties both in *Gnutella* and *Catastrophic*.

Our tests use the following default parameters. The maxDegree limit is set to 5, whereas minDegree, when used, is set to 2. Nodes can retry connecting to a previously contacted candidate that was *busy* searching for its own parent a maximum of 3 times. The length of the Ancestor Chain is 3 and the probability of choosing a regional candidate from the Ancestor Chain or the Sibling Set is 0.5. We experimented with other values, however the use of a longer Ancestor Chain or of different probabilities did not significantly affect performance. The Global cache contains at most 10 entries, while the number of references in refusal messages used to update the Downstream cache is 3. Nodes periodically purge their Global caches from failed nodes by sending ping messages every 250s, and they update its content by exchanging messages with their neighbors with the same interval. We chose these values to strike a balance between the sizes of caches and messages and the need to reconnect the overlay effectively.

Part of our analysis was devoted to identifying the combinations of repair strategies that provide the best performance in terms of our protocol's soft re-

[2]We gratefully thank Stefan Saroiu and Krishna Gummadi for providing the data trace.

Abbreviation	Sequence of caches
RMG	Regional, BreakMaxDegree, Global
RDGM	Regional, Downstream,Global, BreakMaxDegree
DRGM	Downstream, Regional, Global, BreakMaxDegree
RUmDGM	Regional, Upstream, BreakMinDegree, Downstream, Global, BreakMaxDegree
DUmRGM	Downstream, Upstream, BreakMinDegree, Regional, Global, then BreakMaxDegree
Overcast+G	Ancestor Chain, Global— no degree constraints

Table 10.2: Combinations of recovery strategies. Strategies without BreakMinDegree do not enforce minDegree.

quirements. This has led to the selection of the protocol instances summarized in Table 10.2. The first three involve only a subset of repair strategies and evaluate our LSTree protocol without the minDegree limit (but with maxDegree). The next two, instead, enforce the minDegree limit and also exploit the Upstream and BreakMinDegree strategies. Finally Overcast+G simulates a similar approach found in the literature [59]. Overcast is a system for reliable multicast that maintains an overlay tree using a strategy similar to our Ancestor Chain and with no node degree constraints. Because Overcast alone was unable to maintain a connected overlay in the scenarios we evaluated, we augmented it with our Global strategy.

10.4.2 Results

Our simulation results confirm the ability of our LSTree protocol to keep the overlay connected, while effectively managing node degree and limiting the extent of topological changes induced by failures. In addition they highlight its efficiency both in terms of the time it requires to repair failures and in terms of its communication cost. In the following, we discuss each of these aspects, and highlight how each repair strategy contributes to LSTree's performance.

Ability to Recover Failures. The key feature of a tree maintenance protocol is the ability to maintain the overlay network as an acyclic, connected structure. Our first simulation results confirm that these goals are met in both the *Gnutella* and *Catastrophic* scenarios. To gain a better understanding of the failure recovery process, we analyzed how reconnection is achieved by each of the considered protocol instances. In particular, we measured the percentage of failures repaired with each of the strategies described in Section 10.3.

Both plots in Figure 10.3 show that all protocol instances recover the majority of failures using the Regional strategy, whereas the use of the Global strategy is very infrequent. This is a very positive result because the Regional strategy represents the best option both in terms of delay to reconnect and in terms of locality of the reconfiguration process, as we show in the following sections.

More specifically, in the *Gnutella* scenario (left plot of Figure 10.3), all the selected protocol instances are able to recover all failures. Moreover Global is used in less than 0.01% of the cases. This is reasonable because in the instances shown in Figure 10.3, the Global strategy is only exploited as one of the last repair options. We also experimented with instances in which Global is activated imme-

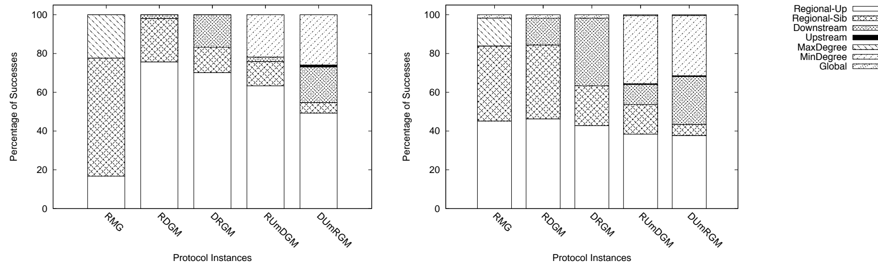


Figure 10.3: Percentage of reconnections achieved by each strategy with several protocol instances in the *Gnutella* (left) and *Catastrophic* (right) scenarios.

diately after Regional and even then its use remains limited without significantly hampering the ability of the protocol to repair failures locally.

The use of the Global strategy is instead more important in the *Catastrophic* scenario. The use of this strategy allows the LSTree protocol instances to respond to the failure of one quarter of the network, restoring a connected topology in all but a few simulation runs. RMG split the overlay into two subtrees in only 2 out of 10 *Catastrophic* runs, whereas the other instances resulted into two separate trees in only 1 *Catastrophic* run out of 10.[3] Moreover, these results are obtained with a fairly small Global cache containing only 10 references in a network of 10,000 nodes. Larger caches and more aggressive update mechanisms can significantly improve these performance figures.

The ability of our LSTree protocol to reconnect the overlay is in sharp contrast with the results obtained by the Overcast instances we simulated. The Overcast protocol alone (not shown in the figure) was unable to achieve reconnection even in the *Gnutella* scenario. An average run resulted in the creation of as many as 430 trees containing a total of 1400 nodes at the end of 60 hours of simulated time. When augmented with our Global cache, "Overcast+G" achieved reconnection in *Gnutella*, but resulted in either two or three separate trees in 5 out of 10 runs of the *Catastrophic* scenario.

Evaluating Node Degree. Next we analyze the ability of the LSTree protocol to control node degree. For this, Figure 10.4 shows the distribution of node-degrees at the end of a sequence of reconfigurations in both of the considered scenarios for our reference protocol instances as well as "Overcast+G".

From this plot, it can clearly be seen that although "Overcast+G" is able to achieve reconnection, it does so at the cost of very high node degrees. The node with the highest degree has an average (over all simulation runs) of 675 neighbors in the *Gnutella* scenario and 99 in *Catastrophic.*

On the other hand, all of our protocol instances effectively manage node degree while keeping the overlay connected at all times. Even RMG, which forces the MAXDEGREE limit to be broken early in the recovery process without even attempting to use the Downstream strategy, yields a degree distribution in which only small fraction of nodes exceed the degree limit and in which the top degree node has an average of 20.8 neighbors in *Gnutella* and 18 in *Catastrophic*. These are reasonable values for most overlay applications.

[3]None of the instances exhibited partitions in any of the *Gnutella* runs.

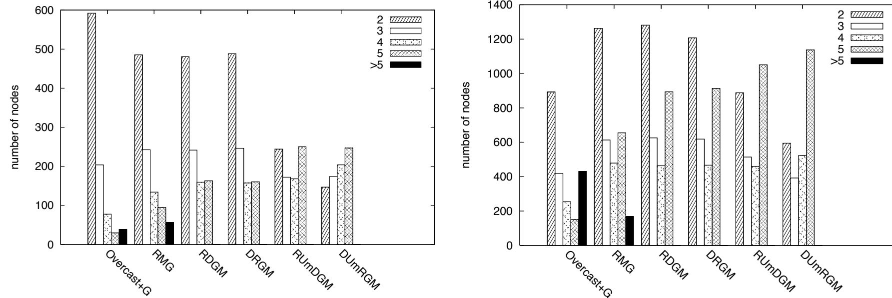


Figure 10.4: Degree of highest-degree node and degree distributions with several protocol instances in the Gnutella (left) and Catastrophic (right) scenarios.

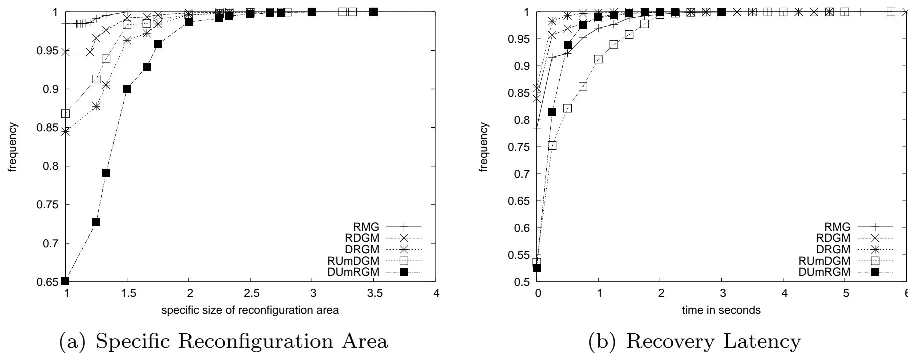
Degree distribution improves further for the RDGM and DRGM instances, which exploit the Downstream strategy and postpone BreakMaxDegree until after Global. Downstream and Global together allow these instances to always achieve reconnection without exceeding the degree limit in *Gnutella* and exceeding it in 1 of 10 runs in *Catastrophic* (an average of 5.1 in the figure).

Keeping node degree below the MAXDEGREE limit is, however, only half of the picture. As we discussed in Section 10.3.1, LSTree also has the ability to shift the degree distribution towards nodes with full degree (i.e. close to MAXDEGREE) by enforcing the MINDEGREE constraint. This has the beneficial effect of reducing the average distance between nodes in the overlay, thereby reducing the latency incurred by messages forwarded along the tree.

The two rightmost protocol instances in Figure 10.4 show the results obtained when the MINDEGREE limit is enforced. In both cases, degree distribution significantly improves, reducing the number of nodes with only two neighbors in both the *Gnutella* and *Catastrophic* scenarios. Moreover, DUmRGM achieves a better degree-distribution than RUmDGM; this is due to its more frequent use of the Upstream strategy as evidenced by Figure 10.3.

Finally, as another point of comparison, we tested protocol instances using integer-valued depth instead of real numbers. As expected, results show that the introduction of real-numbers significantly improves the management of node degrees. With *integer depth*, DUmRGM reaches a top degree of 9 (instead of the 5 obtained with real numbers), while RMG behaves almost as Overcast+G and reaches a top degree of 520 (instead of the 20.8 reached with real numbers).

Recovery Locality. The third aspect we examine is the area of the overlay affected by a reconfiguration. Specifically, we consider a *reconfiguration area* defined as the union of all the paths in the new tree from all children of a failed node to their former grandparents. Smaller area implies better locality. The relevance of this measure comes from the application of LSTree in our middleware architecture. The reconfiguration area is an extension of the reconfiguration path defined in Section 5.2 and represents the area in which brokers in a publish-subscribe system need to update their routing information to adapt to the new topology. Moreover, the metric is general enough to give an idea of the impact of topology changes on other types of middleware and applications. Because the size of the area depends



(a) Specific Reconfiguration Area (b) Recovery Latency

Figure 10.5: Cumulative distributions of the time to recover a failure and of the size of the reconfiguration area with *Gnutella*.

on the number of children of the failed node, we divide the size by this number, obtaining a specific reconfiguration-area size.

Results depicted in Figure 10.5(a) highlight LSTree's ability of to localize changes. In four out of five protocol instances, over 85% of recoveries have a specific reconfiguration-area size of 1, meaning nodes always find a new parent among the neighbors of a failed node. More precisely, RMG yields the best performance since nodes are almost always able to reconnect to their grandparents or to their siblings at the cost of increasing their node degree. On the other hand, RDGM, DRGM, and RUmDGM effectively keep all nodes within the degree limit with only slightly lower locality. For RDGM, the size of the specific reconfiguration area is 1 in 95% of the cases, while for DRGM and RUmDGM it is smaller than 1.5 in 95% of the cases. The curves associated to these last two instances are almost equivalent, albeit for two different reasons. In the case of DRGM, the Downstream strategy is given a higher priority than Regional. This means that as soon as a Regional node refuses a request because of its MAXDEGREE limit, the search for the new parent continues downstream and thus farther away from the failed node. In the case of RUmDGM, on the other hand, the reason for the decreased locality is related to the presence of the MINDEGREE limit. The *relative worst* performance among the five selected instances is achieved by DUmRGM, where the non-local effects associated to the Downstream strategy and the MINDEGREE limit combine with each other. Nevertheless, its specific reconfiguration area is still smaller than 2 in 95% of the cases. Finally, as in the case of node degree, we tested the performance of instances based on integer depth. Results show that the use of real numbers also improves the performance of LSTree with respect to locality: DUmRGM recovers failures with specific reconfiguration area of 1 in 65% of the cases with real numbers but only in 50% of the cases with integer depth.

Recovery Delay. Next we evaluate the time required to reconnect the overlay after the failure of one or more nodes. This is measured as the time required by the children of failed nodes to locate and connect to new parents.

Figure 10.5(b) shows the cumulative distribution of recovery delays obtained in *Gnutella*. Results show that the protocol is able to recover failures promptly.

In four out of five protocol instances, 95% of recoveries complete within 1 second, whereas RUmDGM still manages to recover the same in under 2 seconds.

Interestingly the best performance in terms of latency is achieved by DRMG. The use of the Downstream strategy provides a quick and efficient way to react to connection refusals determined by the maxDegree limit. More precisely with DRMG all repairs require only a few message exchanges and complete within 1 second in the worst case. Only slightly worse is the performance of RDMG: in this case the Downstream strategy is activated only after all regional candidates have been contacted, yielding better locality at the cost of a longer repair process.

Similar observations can be made for the RMG strategy. In this case the protocol does not look for downstream candidates; yet, it still exhausts the Regional strategy before attempting to override the degree limit. Moreover, the use of BreakMaxDegree increases the chances that nodes remain over maxDegree, thereby refusing subsequent connection attempts issued with the Regional strategy. This results in the higher latency exhibited by RMG in Figure 10.5(b).

The worst performance in terms of latency is exhibited by the RUmDGM instance, which nevertheless takes at most 3.5 seconds. This is due primarily to the minDegree limit. Together with the Upstream and BreakMinDegree strategies, this limit reduces the number of 2-degree nodes at the expense of a less local and longer reconfiguration process. It is worth observing that the cost associated to the use of the minDegree limit depends on the placement of the BreakMinDegree strategy in the priority sequence: the lower the priority, the longer and less local the recovery process. To confirm this, we also experimented with other protocol instances (RDUmGM and RDGUmM) which waited even longer before overriding the minDegree limit. These instances show longer time to recovery, but improve the distribution of node degrees with respect to RUmDGM.

Our LSTree protocol, however, is also able to improve degree distribution without increasing latency. In the previous discussion, we noted that the DUmRGM instance achieves very good performance in terms of node degree, and Figure 10.5(b) shows that its latency is comparable to that of instances that do not enforce minDegree. The reason is that the repairs with a high latency caused by the presence of the minDegree limit are balanced by the quick repairs resulting from the high priority given to the Downstream strategy.

The above considerations suggest that DUmRGM is likely to have the widest applicability because it effectively balances a very good degree distribution and a quick reconfiguration with a specific reconfiguration area which, albeit not as small as with other instances, is smaller than 3 hops in practically all cases.

These results are confirmed in the *Catastrophic* scenario: the latency of all instances increases slightly, but DUmRGM still achieves its good degree distribution and recovers 95% of failures within 3 seconds, with a specific reconfiguration area smaller than 3 hops in all cases.

Finally, it is worth noting that the use of real-number depth is again beneficial for our LSTree protocol as instances based on integer depth generally result in longer reconfiguration delays. In particular, in *Gnutella*, DUmRGM recovers 95% of failures within 1 second with real-number depths, whereas it requires almost 2 seconds when integers are used.

Cost of Tree Maintenance. The final piece of our performance evaluation ad-

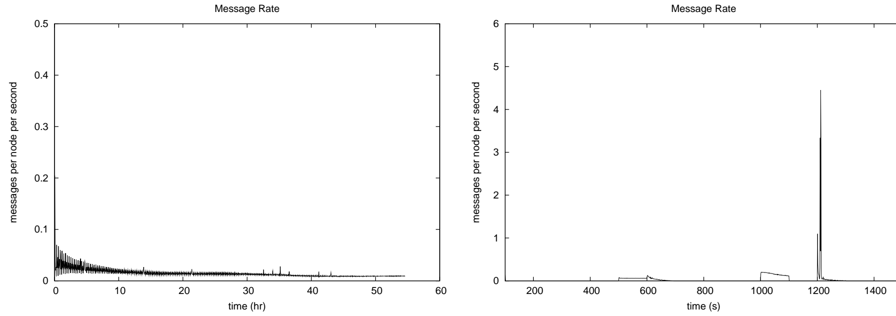


Figure 10.6: Number of messages per node per second in the Gnutella (left) and Catastrophic (right) scenarios.

dresses cost in terms of messages per node per second. We include beacons used to purge failed nodes from the Global cache, but exclude those used to check for failed neighbors since these are redundant when neighboring nodes exchange application-level traffic.

Results show that LSTree is efficient in all the instances we evaluated. Figure 10.6 presents those obtained with RMG, but equivalent results are obtained with the other instances. The left plot in the figure shows that in *Gnutella*, the traffic generated by LSTree always remains well below 0.1 messages per node per second, with only minor fluctuations, determined by fluctuations in the frequency of node connections and disconnections.

In the *Catastrophic* scenario, shown in the right plot of the figure, the result is even more interesting. In a stable topology, the protocol requires less than 0.01 messages per node per second for refreshing the contents of Global caches. When the failure of 2500 nodes occurs, the number of messages exhibits a spike. However, even at its peak, traffic remains below 5 messages per node per second.

These results are particularly interesting because they represent a significant

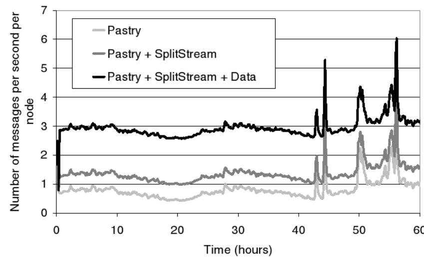


Figure 18: Number of messages per second per node with the Gnutella trace on GATech.

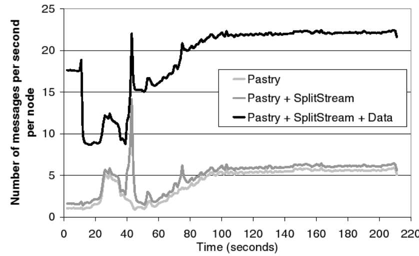


Figure 16: Number of messages per second per node when 25% out of 10,000 nodes fail on GATech.

Figure 10.7: Results obtained by the protocol in [21], in situations resembling our Gnutella (left) and Catastrophic (right) scenarios.

improvement with respect to protocols that build trees as subsets of more connected mesh overlays. As a term of comparison, Figure 10.7 shows two plots taken from a paper describing SplitStream [21], an application-level multicast system built over the mesh overlay provided by the Pastry [63] DHT. The results shown in the left plot were obtained using the same data trace as the one we used in our *Gnutella* scenario, while those in the right plot were obtained in a network of 10,000 nodes, in which 2500 nodes crash after 10 seconds of simulation similarly to our *Catastrophic* scenario.

The plots from [21] show that the maintenance cost for the Pastry overlay is generally higher than that associated to the maintenance of our own (Figure 10.6). Moreover, the right plot shows that the traffic generated by the Pastry overlay takes fairly long to stabilize. Nodes crash after 10 seconds of simulation, and after 200 seconds, traffic is still higher than 5 messages per node per second, including beacons. Our protocols on the other hand stabilize much more quickly.

The absolute numbers cannot be compared exactly because Figure 10.6 assumes a lower beacon interval for the LSTree protocol (failures are detected within 150ms-200ms) and does not count the cost of beacons. On the other hand, Figure 10.7 includes the cost of the beacons sent by Pastry every 30 seconds in the absence of application level traffic. Although using such a beacon interval in our LSTree protocol would delay the detection of failures, it would not increase stabilization time. Moreover, adding the communication cost of beacons would increase LSTree's overhead by a constant factor, bringing its cost close to the overhead of Pastry before the crash. However, LSTree returns quickly to this initial cost, staying well below the 5 messages per node per second incurred by Pastry following the crash and before stabilization.

10.5 Concluding Remarks

Overlay trees are among the most common topologies for data distribution systems designed for large-scale networks. However they have often been regarded as unable to deal with dynamic environments due to their inherent fragility. In this chapter, we confronted this assumption with a novel protocol for maintaining a tree overlay in a highly dynamic environment while at the same time managing node degree and limiting the impact of changes in the overlay. Our simulation results demonstrate the effectiveness of the LSTree protocol to achieve these goals through a quick and communication efficient repair process.

The work presented in this chapter is also open to future developments. Our most immediate plans include extending the protocol with additional soft constraints and repair strategies. Soft constraints may take into account bandwidth or communication latency between nodes, thereby obtaining trees that have a closer match with the underlying physical topology. Repair strategies may instead consider application-specific forms of locality: in the case of publish-subscribe this would allow the overlay to group nodes with similar interests thereby improving the overall performance of the middleware.

Finally, we also aim to apply the work outside the large-scale, wired domain, exploring the possibility to use ideas such as real-number depth to improve wireless multicast protocols.

CHAPTER
ELEVEN

A DHT-based Overlay

In this chapter we present a different approach to the maintenance of a tree overlay topology. In the LSTree protocol we addressed at the same time the problem of maintaining a connected topology and that of satisfying the requirements of the middleware on its top. In the protocol presented in this chapter, DHTree, we separate these two concerns by exploiting a Distributed Hash Table to replicate the characteristics of a given reference topology on the overlay network. Our DHTree protocol, whose original description can be found in [32], supports arbitrary tree topologies, and the use of a DHT allows it to deal with the dynamicity of network scenarios with only a limited impact on the routing reconfiguration carried out on top of the overlay.

These results are confirmed by simulations that validate the applicability of our approach in reconfigurable publish-subscribe middleware. As with the LSTree protocol, however, applicability goes well beyond the context of content-based publish-subscribe, as the protocol provides a general way to maintain an overlay network with a controlled topology in dynamic environments.

The chapter is structured as follows. Section 11.1 introduces distributed hash tables. Section 11.2 describes the DHTree protocol, while Section 11.3 complements this description with an evaluation through simulations.

11.1 Distributed Hash Tables

Our DHTree protocol addresses the requirements described in Chapter 9 by maintaining a tree topology with minimal impact on the protocols responsible for rearranging subscription information. At the basis of this approach, is the use of Distributed Hash Tables [89, 99, 83, 82].

These systems offer a distributed lookup functionality in large scale network environments. Like a traditional hash table, they provide a mapping function between a set of keys and a set of stored objects. The distribution aspect lies in the fact that objects are stored at various locations in the network.

Distributed hash table implement the mapping between keys and objects by first mapping each key to a host that is designed as its owner. To accomplish this hosts are also associated to a hashed identifier value. Moreover, a distance function is defined over the set of hashed values. An object's key is assigned to the host whose hashed identifier is closest according to the defined distance function to hashed key value. This process, known as key-space partitioning, yields a possibly uniform distribution of keys (and hence of objects) among the hosts in the network.

DHTs manage this distribution of key values with an efficient lookup operation. The hosts in a DHT are organized in an overlay network consisting of a mesh topology. The fundamental property enabling routing in this topology is that given a key k, each host either *owns* k or it can identify one of its neighbors which is closer to k according to the predefined distance. This yields a lookup protocol characterized by a communication complexity on the order of the logarithm of the number of hosts. Moreover, the use of a mesh topology allows this protocols to operate without significant performance problems even in scenarios characterized by host failures and disconnections.

Several DHTs have been presented in recent years, the most relevant being Chord [89], Tapestry [99], Pastry [83], and CAN [82].

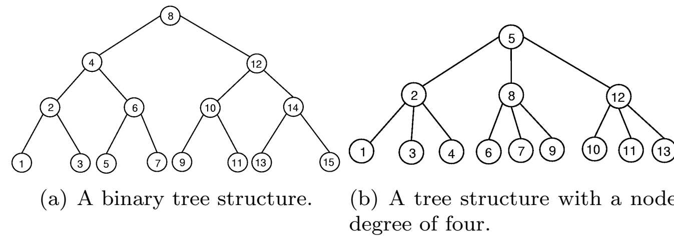
Chord is a DHT based on a ring network topology augmented with "shortcuts" between hosts that are a-power-of-2 hops apart in the ring. Hosts in the ring are ordered according to the values of their identifiers. Moreover, each host is designated as the owner of those keys whose values are greater than or equal to its own identifier, but less than that of the peer immediately following along the circle.

Pastry and Tapestry, on the other hand organize the hosts in a hypercube-like topology. Keys and host identifiers are managed as sequences of n digits of m bits. Each messages is routed between host so that the host reached at the nth hop has an identifier sharing the first n digits with the message destination key.

The Content Addressable Network (CAN) takes a different approach and organizes keys and hosts in an n-dimensional cartesian space. Each host is assigned an n-dimensional area of this search space. When the first host enters in the system, it is initially assigned the entire space. Subsequently, when new a host joins, it contacts an existing host and receives a portion of its own area. Keys are associated to a location in this cartesian space; messages for a given key are then forwarded towards the host that owns the area containing the key's location.

11.2 Protocol Description

The flexibility of distributed hash tables, combined with their ability to withstand topological changes such as node failures, motivate the design and development of an overlay maintenance protocol for reconfigurable publish-subscribe systems on their top. In the rest of this chapter, we present the results of our efforts in this direction: DHTree, a protocol that exploits a DHT to build and maintain a tree-shaped overlay network satisfying the requirements of publish-subscribe middleware.



(a) A binary tree structure.

(b) A tree structure with a node degree of four.

Figure 11.1:

The protocol is based on a very simple idea. We construct a predefined reference tree structure and distribute it to the hosts in the network, either before the system is deployed or exploiting the routing abilities of the DHT. This predefined tree structure is simply a picture of what the desired overlay topology should look like. Our DHTree protocol exploits the DHT to translate this reference topology into an actual overlay network.

In the following we assume the use of DHT satisfying the following two properties on the mapping between keys and hosts. First *each key must be assigned to one and only one host*. Second, *each host must be aware of the keys it has been assigned*. The first is a very basic property satisfied by all DHTs whenever their data replication mechanisms are not considered. The second only requires the interface of the DHT to expose to each host the information about the keys in its own keyset and can therefore be easily implemented even in DHTs that do not natively offer this functionality.

These two properties allow us to implement a mapping between the hosts in the network and the nodes in the reference topology. First the DHT maps a host onto its set of available keys; then our DHTree protocol maps this set of keys onto a specific node in the reference tree structure.

To simplify the description of the protocol we will try to avoid confusion between the terms referring to the elements of the reference tree structure and to those referring to the overlay. Specifically, we will use the term *node* to refer to the nodes in the common tree structure and the term *host* to refer to the elements of the overlay. In addition we will use the term *key* to refer to the possible identifier values stored by the DHT.

11.2.1 Reference Tree Structure

Figure 11.1(a) depicts an example of a binary reference tree structure. Each node in the structure is associated to a numerical value taken from the set of all possible key values. The size of the structure is configured so that it covers the entire keyset managed by the DHT. A consequence of this fact is that the reference tree structure must have at least as many nodes as the number of hosts participating in the overlay.

Hosts operate by associating the keys in this reference structure to the keys they own according to the keyspace partitioning scheme implemented by the DHT. More precisely, each host chooses a representative node in the structure in the

following way. It first obtains its own set of keys by querying the DHT; then it uses this *keyset* to select a single node in the reference tree. Specifically it traverses the reference tree in breadth-first order until it reaches a node that is labelled with one of the keys in the *keyset*.

This can be summarized by the following definitions:

Definition 1 *Let h be a host, we define the* keyset *of h, K_h, as the set of keys assigned to h by the DHT.*

Definition 2 *Let K be a set of keys, the* topmost *key in K, $\overline{K}$, is the first key value in K encountered in a breadth first traversal of the reference tree structure, starting from the root node.*
We also define the topmost *key $\overline{h}$ of a host h to be the topmost key of its keyset K_h, that is $\overline{h} = \overline{K_h}$.*

This concept of topmost key is what enables the mapping between hosts and the reference tree structure. Each host is associated to the node in the reference tree structure labelled with its own topmost key.

11.2.2 Determining a Node's Neighborhood

This above process associates each host with a uniquely determined node of the tree structure: its topmost key. Nevertheless, it does not specify how hosts can determine the identity of their neighbors. The general idea is that a host with topmost key k will be the neighbor of another host with topmost key k' if k is a neighbor of k' in the reference tree. In practice, however, each host owns more than a single key: therefore only a subset of the keys in the reference tree can be designated as topmost. For this reason, the neighborhood relation between nodes in the reference structure is not enough to define neighborhood between hosts.

Hosts must therefore use a recursive mechanism to traverse the reference tree structure starting from their own topmost keys until they find a topmost key that can serve as a neighbor. In the following we describe this recursive search process distinguishing the case in which a host is locating its parent from the case in which it is locating its child nodes.

Determining the parent host To facilitate the description of the neighbor search algorithm, it is convenient to define the *parent key* of a topmost key as follows.

Definition 3 *Let k be a topmost key, we define the* parent key *$par(k)$ of k as the closest key to k along the path from k to the root key, such that $par(k)$ is the topmost key of some host and $par(k) \neq k$.*

Exploiting this definition, we can easily define the parent of a host with topmost key k as the owner of $par(k)$. To determine the owner of $par(k)$, however, a host must first determine $par(k)$. To accomplish this, it simply traverses the reference tree structure starting from its topmost key k and continuing towards the root node until it finds a node labelled with a topmost key. More precisely, it first determines the parent node of its topmost key in the reference structure,

k_p. Then it queries the DHT to determine the whether k_p is the topmost key of some host h_p. If this is the case, then $par(k) := k_p$ and h_p is chosen as the parent, otherwise the process is iterated with the host visiting the parent node of k_p.

Determining child hosts The children of a host are determined in a similar way. A host h willing to determine its children first checks whether the child nodes of its topmost key are themselves topmost keys of some hosts. If this is the case, the host takes the corresponding hosts as its neighbors.

In addition, the host carries out a depth-first traversal in each of the subtrees rooted at the child nodes that are not topmost keys of any host. Each of these traversal backtracks either when reaching a leaf node in the reference tree or when reaching the topmost key of some host. This process allows the host to locate the set of all topmost keys k_c such that the path from k_c to the topmost key of the host, $\overline{h}$, contains no other topmost keys. The host collects this set and records the owners of the keys in the set as its children.

Mutual Parent-Child Agreement The two search algorithms outlined above allow a host and its parent to retrieve each other's identities by means of two independent search processes. Moreover, the search for a child host retrieves all the closest topmost keys in the subtree rooted at the host. This guarantees that the search for a parent and the search for children agree in their findings. In other words, if a node n determines that another node m should be its neighbor, then m, in turn, determines that it should connect to n.

Managing Neighborhood Changes The neighbor search process is initiated either by a host detecting the disconnection of one of its neighbors or by a host that joins the overlay for the first time. In both cases the host computes the identities of its parent and child hosts and then compares these identities with those of its former neighbors; none in case the node is joining.

The result of this comparison allows the host to notify a neighborhood change to all the hosts in the symmetric difference of the two sets, that is to all the new neighbors as well as to all the former neighbors that are not in its new neighbor set. Each host receiving a neighborhood change notification executes the neighbor search algorithm. Clearly, no notifications are sent by hosts that do not record changes in their neighbor set, allowing the protocol to terminate in a finite number of steps.

For example, a host joining the network determines which other nodes should become its neighbors and establishes a connection with them. These hosts, in turn, recompute their own sets of neighbors and modify their connections. All the hosts that are added to or removed from some other host's neighbor set are forced to recompute their own neighbor sets and operate accordingly.

This iterative process seems to suggest that the joining or leaving of a single node may modify significantly the interconnections between hosts. However, our empirical evaluation shows, as discussed in Section 11.3.2, that these changes remain confined to a restricted area.

11.2.3 Maintaining Overlay Properties

The protocol described up to this point correctly maintains a tree overlay topology starting from the reference tree structure provided as input. In addition, when the number of hosts equals (or is close to) the number of available keys, the shape of the overlay coincides (or is close to) that of the reference tree structure.

Our purpose, however, is to obtain a topology that is always "close to" that of the reference tree structure, even if the number of hosts is much less than the number of available keys. To achieve this purpose, we establish some additional requirements on the reference tree structure and on the DHT implementation.

Properties of the reference tree structure The first of these requirements is that the assignment of key values to the nodes in the reference tree structure must satisfy the following property.
Let n be a node in the tree structure and p its parent and let $k(n)$ and $k(p)$ denote their keys. Let $k_m = min(k(n), k(p))$ and $k_M = max(k(n), k(p))$, then node n has at most one child node c whose key k_c does not lie in the interval between k_m and k_M.
Additionally, $k_c < k(n)$ if $k(n) < k(p)$ and $k_c > k(n)$ otherwise.

In the case of binary trees (with node degree equal to three), the property is satisfied by a binary search tree like the one shown in Figure 11.1(a). For larger values of node degree, it is possible to define tree structures similar to the one in Figure 11.1(b). It should be noted that satisfying this property is not overly restrictive as it only constrains the way in which key values are mapped on to the nodes in the reference tree. Any reference tree can therefore be used and care need only be taken when assigning key values to its nodes.

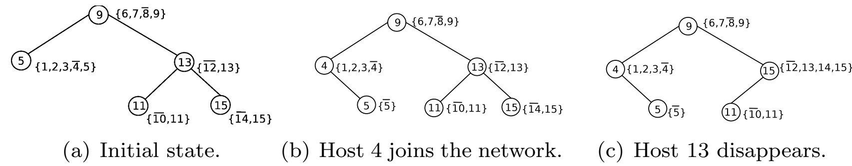
Properties of the DHT The additional requirement on the DHT, on the other hand, specifies how key values must be mapped onto network hosts. Specifically, we require that the DHT satisfy the following property.

- *If k_1 and k_2 are mapped onto the same node n, then either all the keys between k_1 and k_2 or all the keys between k_2 and k_1 are also mapped onto n.*

Chord [89] is probably the DHT which most naturally satisfies this property as well as the previously mentioned requirements. However most other DHTs can be used for this purpose.

Properties of the Overlay The two requirements we just outlined allow our DHTree approach to control the number of neighbors of each host in the overlay network. As mentioned in Chapter 9, this property helps guarantee the scalability of the middleware by preventing resource-constrained hosts from having too many neighbors.

In the case of a binary reference tree structure, DHTree guarantees that the number of neighbors of each host is at most as large as the degree of the nodes in the reference tree structure. In the case of more general reference trees, this property holds with high probability as shown by our simulation results.



(a) Initial state. (b) Host 4 joins the network. (c) Host 13 disappears.

Figure 11.2: Sample reconfigurations with nodes being added and removed.

11.2.4 Example

An example of the behavior of the algorithm is provided in Figure 11.2. The figure shows a reconfiguration with hosts joining and leaving the overlay network.

The reference tree structure for this example is the one depicted in Figure 11.1(a). Let us consider the set of hosts $\{h_5, h_9, h_{11}, h_{13}, h_{15}\}$[1]. Exploiting the service offered by the DHT, each host determines its position in the overlay network autonomously without further communication with the other hosts. This yields the topology depicted in Figure 11.2(a): the network configuration is sketched together with the keyset of each host and its topmost key, shown with a bar on top. Host h_9 discovers it is designated to act as the root since its topmost key $\overline{h_9}$ is equal to 8, which is the root key in the reference tree structure. It then determines its children by querying the DHT for the owners of keys 4 and 12 (hosts h_5 and h_{13}) and creates a link with them.

Analogously, h_5 and h_{13} realize their topmost keys are, respectively, 4 and 12 and that their parent h_p is h_9, since $par(k) = 8$ for both. Besides, h_5 tries to detect its children, starting from its left child. It does not find any topmost key exploring its left subtree (keys 1, 2 and 3 belong to h_5's keyset) and continues by checking whether a right child is available. The first key it encounters is 6, which falls inside K_{h_9}, but is not h_9's topmost key. Therefore, h_5 considers the next keys occurring in depth-first order, namely 5 and 7. Again, these keys are not topmost keys (key 5 is owned by h_5 itself and key 7 is in K_{h_9} but, as described above, $\overline{h_9} = 8$) and so they are discarded. As no other key is available in this sub-tree, h_5 terminates its procedure by connecting only to its parent h_9.

Exploring its sub-tree, h_{13} discovers that keys 10 and 14 are the topmost keys of h_{11} and h_{15} and establishes a connection with them. Similarly, h_{11} and h_{15} connect to h_{13}, the owner of their $par(k)$. None of them connects to other nodes as no topmost keys are present in their sub-trees.

Connection of a Host Now suppose that h_4 decides to join the network: it notifies the DHT of its presence and retrieves its keyset $K_{h_4} = \{1, 2, 3, 4\}$ (note that K_{h_5} changes accordingly and now contains only key 5). Its topmost key is 4 and its parent is the owner of key 8, i.e. h_9. As for its children, it cannot find any left child since keys 1, 2, 3 belong to K_{h_4}, while it uses h_5 as right child, since

[1]For the sake of clarity we adopt the key distribution scheme of Chord [89] according to which each host is responsible for all the keys ranging from its predecessor (excluded) to itself (included). Nevertheless, as mentioned above, all the schemes satisfying the aforementioned properties can also be used.

h_5's topmost key is now 6. When contacted, h_9 and h_5 realize that the topology has changed and consequently check whether they have to modify their neighbor sets. In this case, no further modifications are needed (other than the connection to h_4), so no action is performed. The final result is depicted in Figure 11.2(b). Note that hosts h_{11}, h_{13} and h_{15} are unaffected by the reconfiguration and no computation or message exchange is required by them.

Disconnection of a host Now let us consider what happens when host h_{13} fails. Host h_9 detects a disconnection and updates its neighborhood adopting the usual strategy, finally connecting to h_{15} (whose topmost key is now 12). In turn, h_{15} is notified of the topology change and initiates an update procedure that eventually connects it to h_9 as its parent and to h_{11} as its child. Host h_{11} is also made aware of the topology change and updates its neighbors analogously. This process results in the final topology depicted in Figure 11.2(c).

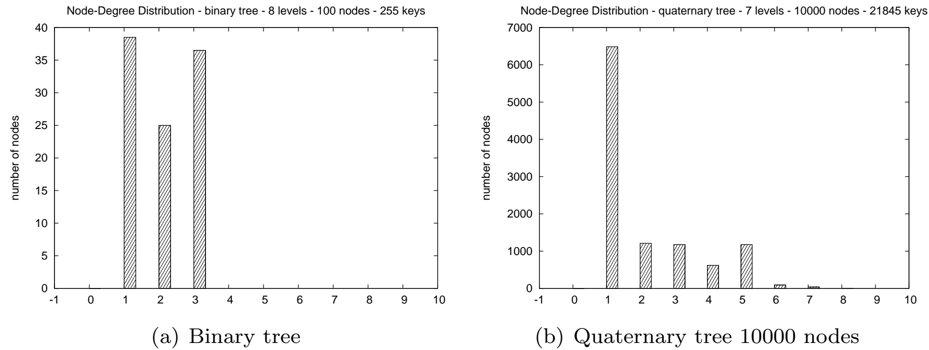
11.3 Evaluation

Similarly to what we did in the previous chapters, we implemented the protocol using OMNet++ [95], a discrete event simulation system. In addition, we developed a test application in Java to improve our understanding of the mapping between key sets and topmost keys. The purpose of our evaluation is twofold. Firstly, they serve as a test to verify that the protocol is indeed able to maintain the desired overlay in a network with hosts that join and/or leave at arbitrary times. Secondly, they provide a measure of its impact on the reconfiguration of routing information in a distributed publish-subscribe system.

11.3.1 Simulation Setting

The scenario we adopted in our simulations consists of a network of hosts which connect and disconnect at random intervals. The number of hosts in the network, N, is kept approximately constant by equating the rates at which connections and disconnections occur. This setting is built incrementally, starting from an empty network and having new hosts join the existing overlay using the protocol described in Section 11.2. As soon as the network reaches a size of N hosts, the system evolves to a state in which each connection is followed by a disconnection after a specified interval $T_{add,remove}$. Measurements are taken for a time interval T_{meas} from the moment in which the network reaches its size.

The DHT is simulated as an abstract component with the ability to map keys onto hosts and conversely. In particular, we assume that the DHT offers the same consistent view to all the hosts participating in our DHTree protocol. This corresponds to DHTree's being invoked only after the DHT has stabilized subsequently to the connection or disconnection of a host. We ran simulations for the binary case using a key-space of 255 identifiers, which also determines the size of the reference tree structure, and an actual number of hosts ranging from 10 to 120. In addition, we ran tests using a quaternary reference tree structure to evaluate the resulting degree distribution.



(a) Binary tree

(b) Quaternary tree 10000 nodes

Figure 11.3: Degree distributions for a binary tree with 100 nodes (left) and for a quaternary tree with 10000 nodes (right)

11.3.2 Results

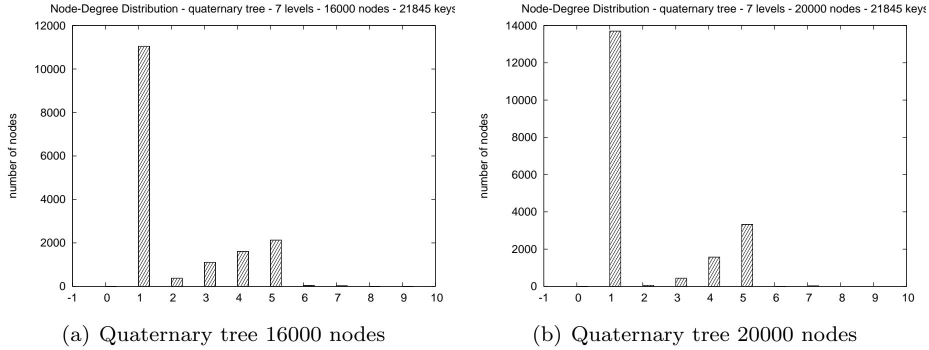
Our experiments suggest that our DHTree approach is a valuable candidate when choosing a topology maintenance algorithm for a distributed publish-subscribe system. Our results are presented as follows. In Section 11.3.2 we evaluate the characteristics of the topology maintained by the algorithm, while in Section 11.3.2 we analyze the impact of the DHTree approach on a publish-subscribe system built on top of it.

Correctness and overlay properties

In order to asses the protocol's ability to maintain the desired overlay network, we ran simulations for a time interval T_{meas} and evaluated the characteristics of the resulting overlay after a period of reconfigurations. More precisely, we first checked that the desired topology was indeed a tree, that is an acyclic and connected graph, and then evaluated the number of neighbors of each host to verify that the resulting node degree was bounded by that of the reference tree structure. As mentioned previously, the bound is strict in the binary case, while it holds with high probability in general.

Figure 11.3 shows the average distribution of node-degree obtained with random sets of nodes both in the binary and quaternary cases. The left plot shows that the degree of nodes never exceeds the value of 2 in the binary case. The right plot instead shows that only a minimal portion exceed the degree of the reference tree structure in the non-binary case even when the number of hosts is only half the number of available keys.

Figure 11.4 shows the same measurements taken on quaternary trees with 16000 and 20000 hosts. The plots clearly show that the resulting topology becomes closer to an ideal balanced topology as the number of hosts approaches the number of available keys.



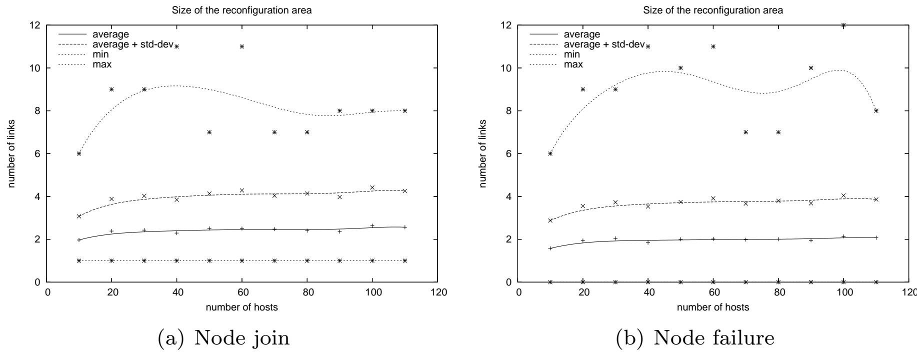
(a) Quaternary tree 16000 nodes (b) Quaternary tree 20000 nodes

Figure 11.4: Degree distributions for a quaternary trees with 16000 (left) and 20000 (right) nodes.

Recovery Locality

The other measurement we consider is the locality of the reconfiguration process resulting from the joining or leaving of a node. Similarly to what we did in Chapter 10, we consider the size of the reconfiguration area by computing the size of the subtree interconnecting the endpoints of the links removed or created as a result of topology changes.

The results obtained in our simulations, depicted in Figure 11.5, show that the size of the reconfiguration area remains very low even when the number of hosts becomes large. The size of the reconfiguration area can be compared with the number of hosts which are directly involved in the topology change. In the ideal case, the reconfiguration area should only consists of the appearing or disappearing host and its neighbors. Although DHTree is not as close to this ideal case as the LSTree protocol, the average number of links added or removed during a



(a) Node join (b) Node failure

Figure 11.5: Size of the reconfiguration area in the case of host connections (top) and disconnections (bottom).

reconfiguration remains between two and three.

11.4 Concluding Remarks

This chapter presented a protocol for maintenance of the overlay tree topology at the basis of distributed publish-subscribe middleware, exploiting the potentialities of distributed hash tables. The protocol exhibits interesting properties such as the ability to control the node degree of the resulting overlay network, and a low impact on the reconfiguration process dictated by topology changes. These properties are validated through simulations that confirm the validity and the potentiality of the proposed approach. Finally, although this thesis describes the protocol in the context of publish-subscribe middleware, this work can be applied to other group communication middleware as well as to any system requiring the maintenance of application-level tree overlays.

The work on this protocol is continuing with further investigation of its characteristics, both from an empirical and from a theoretical point of view. In addition we are analyzing protocol extensions for the maintenance of arbitrary graph topologies.

CHAPTER

TWELVE

Discussion and Related Work

The two overlay protocols presented in the previous chapters exemplify two opposite approaches to the maintenance of an overlay network for publish-subscribe middleware. Both protocols seek to satisfy the requirements posed by the routing layer on the overlay structure as well as on its maintenance process. Nevertheless, their approaches to maintaining these properties are inherently different.

In this chapter, we analyze these differences and we compare our protocols with existing work, highlighting their potentiality not only in the area of content-based publish-subscribe middleware but also in broader contexts such as application-level multicast. The chapter is structured as follows. First we compare the approaches taken by our protocols and highlight their differences both at the interface and implementation levels. Second, we analyze their potentiality in the context of content-based publish-subscribe middleware. Third, we present related solutions both in the context of publish-subscribe and in other application domains; and finally we present some concluding remarks on our results and immediate plans.

12.1 Two opposite approaches

The two protocols we presented differ along two main dimensions. From an interface perspective, each protocol allows the middleware designer to specify the characteristics of the desired topology in different ways. Likewise, at the implementation level, the two protocols use totally different approaches to maintaining these properties.

Specifying Overlay Properties The DHTree protocol focuses on replicating the shape of a reference tree provided as input using a distributed hash table as a mapping function. LSTree, on the other hand, allows the user to specifically

address each overlay property individually with different reconfiguration strategies that can be combined in a single recovery process.

On the one hand, the use of a reference topology provides a simpler interface allowing the overlay to be tuned according to a single topology model. On the other hand, the mechanisms provided by LSTree allow the middleware designer to manage performance tradeoffs explicitly by giving a higher priority to properties that are more relevant to the middleware and application layers deployed on its top.

Moreover, in the DHTree protocol, properties cannot be easily managed on a per-node basis; each network node can in fact be mapped onto different nodes in the reference topology as the system evolves. The LSTree protocol, on the other hand, manages the overlay properties in a completely decentralized fashion guaranteeing better scalability in large dynamic networks consisting of heterogeneous devices.

Property Management Approaches The second set of differences is related to the mechanisms used by the protocols to maintain an overlay with the desired characteristics. The DHTree protocol decouples the management of the overlay properties from the problem of maintaining the overlay in the presence of failures and disconnections. The problem of locating a new neighbor for a node when a former neighbor fails is delegated to the DHT so that different techniques for connection management can be easily integrated by changing the underlying DHT implementation.

The LSTree protocol, on the other hand, sacrifices this decoupling to favor the efficiency of the reconfiguration process. The mechanisms used by nodes to locate new neighbors in case of failures are embodied in the protocol. When nodes look for a new parent, they specifically look for nodes that allow the overlay to satisfy its requirements. This allows the protocol to maintain only the connections that are effectively needed, as opposed to the mesh maintained by a DHT implementation, without sacrificing the ability to maintain connectivity even in the most demanding scenarios.

These differences allow the middleware designer to choose the most effective protocol according to the application scenario. If the middleware is being deployed in a network where a DHT is already operating, the decoupled design that characterizes the DHTree protocol simplifies the implementation of the middleware. Moreover, the protocol allows the DHT and its reconfiguration strategies to be easily shared between different overlays deployed in the same network scenario and not only with other DHT-based applications.

On the other hand, in the case of applications deployed on large scale dynamic environments where the efficiency of communication is relevant, our LSTree protocol provides a more effective topology management mechanism. Moreover, it does not require any type of global common knowledge as is the case with the reference tree structure in DHTree. Simulation results confirm this reasoning and show that the communication cost of the LSTree protocol is even lower than that associated to a DHT [21] operating in the same reconfiguration scenario.

12.2 Overlay in Publish Subscribe

Similar considerations hold with respect to the combination of these protocols with the routing reconfiguration techniques presented in Part II. As we discussed in Chapter 8, each of the protocol poses different requirements on the overlay maintenance layer. At one extreme, TIMED DEFERRED UNSUBSCRIPTION enjoys the widest applicability, while, at the opposite side of the spectrum, RECONFIGURATION PATH is only able to operate if the overlay layer provides information about the entire reconfiguration path.

The overlay protocols we just described respond in different ways to these requirements. The DHTree protocol reacts to disconnections and new connections by having each node recompute its set of neighbors on the basis of the mapping provided by the DHT. Consequently, the endpoints of broken links have no general and easy way to determine which nodes are the endpoints of corresponding new links. In the LSTree protocol, on the other hand, each downstream neighbor of a failed node repairs the overlay by explicitly searching for a new neighbor and therefore for a new link to replace the one that was lost.

The first observation we can make is that the TIMED DEFERRED UNSUBSCRIPTION protocol can be used directly on top of both overlay managers without modifications. In this protocol, network nodes only need to be notified of the appearance or disappearance of communication links and do not need to know which link is replacing which other.

The NOTIFIED DEFERRED UNSUBSCRIPTION and INFORMED LINK ACTIVATION protocols, on the other hand, must be able to make this association between appearing and disappearing links in order to propagate their ACTIVATE messages. While this requirement poses some problems in the case of DHTree, it is very easy to satisfy using the LSTree protocol. The reason is that, in the LSTree protocol, each downstream neighbor of a failed node acts as an endpoint of both the new and the old links, resulting in a simplification of the two protocols. In Part IV we return to this problem in greater detail when describing the integration of our routing reconfiguration protocols with the LSTree layer.

The last observation worth making regards the inherent difficulty of integrating the RECONFIGURATION PATH protocol with either of our overlay managers. This is not unexpected: satisfying the requirements of the RECONFIGURATION PATH protocol would complicate the design and implementation of the overlay layer and consequently limit its applicability. Moreover, RECONFIGURATION PATH is unable to operate correctly in the presence of arbitrarily overlapping reconfigurations, a situation that may well occur in a highly dynamic peer-to-peer setting. Nevertheless, it is worth noting that the inability to exploit the RECONFIGURATION PATH protocol does not hamper our goal to build a highly reconfigurable event dispatching framework. The optimal performance of this protocol can in fact be obtained using the novel INFORMED LINK ACTIVATION protocol, which poses significantly looser requirements on the overlay layer.

12.3 Related Approaches

Together, the considerations we made encourage the integration of these overlay protocols with the routing reconfiguration protocols described in Part II. Nevertheless, the applicability of LSTree and DHTree is not limited to the context of content-based publish-subscribe. Both overlays can be exploited in all large-scale network contexts where a tree overlay network is needed, as is the case in application-level multicast protocols. In addition, some of the techniques from the LSTree protocol can be used outside the wired domain: ideas like real-number depth can be used to improve wireless multicast protocols in mobile ad hoc and sensor network scenarios.

To gain a better understanding of these further application areas, as well as to compare our protocols with existing systems, we provide a brief description of the main solutions for tree-based overlay maintenance in three different contexts: application-level multicast, multicast for mobile ad hoc networks, and publish-subscribe systems. Finally, we also briefly consider theoretical work on the subject and discuss its relationship with our overlay protocols.

12.3.1 Application-Level Multicast

Research in the area of application level multicast has often investigated the problem building efficient data distribution overlays in large scale systems.

One of the first application-level multicast protocols, Narada [30] exploits a two-phase mechanism to build a tree over a mesh structure. Differently from both our protocols, however, Narada is most suited for small groups of machines as it assumes that all nodes in the mesh know about each other.

Bayeux [102], SCAN [28], Scribe [22, 23], SplitStream [21], and I3 [61] are instead related to our DHTree protocol in that they construct data distribution trees on top of DHT infrastructures [99, 83, 89]. Bayeux requires a rendezvous node (root) to handle all join requests by new group members. Scribe uses a more scalable approach and controls the degree of nodes using a mechanism similar to our downstream cache, while SplitStream extends Scribe's behavior to manage the overall degree resulting from a node's participation in multiple trees. Finally, I3 exploits a distributed algorithm to build trees with controlled node degree and low latency.

The way these protocols exploit the DHT is however very different from what we do in our DHTree protocol. First, they tie their resulting topology to the mesh structure constructed by the DHT: this implies that the resulting topology depends on the specific DHT they rely on. Second, approaches based on a rendezvous node require all the hosts in the network to reconfigure when the rendezvous node fails. Moreover, we believe that the optimizations employed by some of these DHT-based approaches are more meaningful in a context without DHT as in our LSTree protocol. Evaluations of these protocols [21] show that the DHT plays a significant role in the overall cost of topology maintenance. Our LSTree protocol, on the other hand, achieve the same level of resilience to failures by exploiting simple structures such as the Regional and Global caches without relying on a separate protocol to manage references to other nodes. This results

in a reduction of the overall cost of the protocol as shown by our simulation study.

Protocols which are more closely related to LSTree are Yoid [48] and Overcast [59]. Yoid reacts to failures by creating separate trees which are then rejoined using a complex distributed cycle-detection mechanism, while Overcast builds an overlay out of a set of dedicated servers and exploits a mechanism similar to our Ancestor Chain. Distributed cycle detection naturally increases the cost of the recovery process, while the approach taken by Overcast causes the topology to reach very high node degrees. Moreover, Overcast is designed for fairly stable conditions, not involving the unexpected failure of the root or of a cluster of nodes, whereas our protocols, particularly LSTree, build an overlay out of hosts exhibiting very dynamic behaviors.

Some other systems, such as CoopNet [72], PeerCast [13], and NICE [12], use a rendezvous node to coordinate the construction of data distribution trees for streaming applications. CoopNet exploits a server to build balanced distribution trees using something resembling our Downstream technique. The server chooses the parents of new nodes by traversing the tree until it finds a node with enough spare capacity. PeerCast adopts a similar approach: nodes first contact the media source and then proceed downstream along the tree until they find an unsaturated member. Albeit interesting, these approaches share the main disadvantage of rendezvous-based solutions based on a DHT-tree. In both cases, the need for a well-known node to handle connection requests by clients makes these protocols unsuitable for systems characterized by a large number of data sources and receivers as is the case in content-based publish-subscribe.

12.3.2 Mobile Ad Hoc Multicast

Establishing multicast communication in a mobile ad hoc network poses similar challenges to building a resilient overlay due to the transient connectivity resulting from the mobility of nodes. The approach that bears the closest resemblance to the domain of overlay networks and to our LSTree protocol is MAODV [84].

As already discussed, one of the main core differences is the use in MAODV of integers to indicate hop count from the root. Because LSTree is especially designed for large scale systems, we avoid flooding the network with depth updates by representing the depth of each node as a real number that is kept consistent as the tree changes.

In addition, MAODV operates in a mobile ad hoc network environment and does not rely on an underlying IP-routing layer. As a result, it is restricted by the physical topology to select links that are currently active. Therefore, instead of caching other nodes in the network, it broadcasts a message to find candidate links.

12.3.3 Overlays for Publish-Subscribe Systems

The final piece of related work we consider is specifically related to the content-based publish-subscribe domain. In addition to our efforts, other research groups have investigated the problem of building publish-subscribe middleware in dynamic environments.

The work in [8] proposes a tree maintenance protocol for publish-subscribe systems that bears some similarities to our LSTree protocol. Specifically, they do not exploit any underlying mesh-based overlays such as DHTs and maintain the overlay connected by having nodes cache the identities of their "grand parents" in the overlay tree. However, while both of our protocols are designed to operate in networks with frequent failures of an arbitrary number of nodes, the authors of [8] assume a failure model with at most one node failure at a time.

The work in [67], on the other hand, is an overlay for mobile publish-subscribe middleware developed in our research group. The protocol is an extension of the MAODV protocol mentioned above, and improves its operation with mechanisms to maximize its performance in the context of content-based publish-subscribe. A brief description of this protocol is provided in Part IV, where we present its integration with the routing reconfiguration protocols developed in this thesis.

12.3.4 Distributed Tree and Spanning Tree Construction

Distributed algorithms for spanning tree construction and maintenance have also been studied from the theoretical perspective. Perlman [75] describes an algorithm used IEEE 802.1 that supports bridge coordination to compute a spanning tree. The protocol exploits a self-stabilizing approach based on periodic broadcast `HELLO` messages that allow nodes to choose their designated neighbors according to the current state of the network. This would be impractical in our case, due to the overhead generated by broadcast messages and because of the absence of control over the extent of topological changes.

The same algorithm [75] was later used as a basis for the development of an algorithm [66] designed for IEEE 1394.1. This algorithm removes the need for periodic beaconing messages and addresses failures by using a node cache, or by having nodes elect themselves as roots of subtrees that will later be joined. This approach does not handle locality, and can therefore cause far reaching modifications in the overlay topology.

Self stabilizing algorithms have also been studied for maintaining a spanning tree in a dynamic network [42, 31]. In these solutions, nodes exchange information about each other and use this information to agree on a tree topology that satisfies some constraints. When the state of the network changes, the nodes agree on a new topology which can be completely different from the previous one. Two major differences exist from the requirements we set for our work. First, Our work targets large scale networks in which it would be infeasible to have each node know about all other nodes in the network. Second, we specifically try to perform local reconfigurations, avoiding far-reaching changes in the overlay information.

Another algorithm [1] exploits the ordering among node identifiers to construct the spanning tree. Again, this algorithm is better suited to local-area or mobile ad hoc networks because it relies on the ability of nodes to detect the appearance of new nodes in the vicinity. Other approaches [6, 26, 56, 4] are also based on the presence of node identifiers but rely on a bound on the number of nodes in the network to detect cycles and operate correctly.

Finally, existing work has also carefully studied distributed algorithms for the

construction and maintenance of shortest path trees [50, 81]. These algorithms, however, often require precise information about the underlying network topology and are not designed to localize topology changes. Moreover they also introduce the possibility of creating temporary routing loops. Our protocols, instead, operates at a higher level and avoid cycles at all times without requiring precise knowledge of the underlying network topology. Moreover, such network knowledge can easily be integrated in our LSTree protocol with new soft constraints concerning path latency, bandwidth, etc.

12.4 Concluding Remarks

Overlay trees are among the most common topologies for data distribution systems designed for large-scale networks. These overlays are often utilized in dynamic environments that make it difficult to keep the nodes of a tree connected. This part of the thesis addressed this problem with two protocols for maintaining a tree overlay in dynamic environments. Our simulation results demonstrate the effectiveness of both protocols to achieve this goal. Moreover they highlight the efficiency of the LSTree protocol in managing node degree and keeping reconfiguration localized when possible through a quick and communication-efficient repair process.

Part IV

Putting It All Together

CHAPTER

THIRTEEN

Integration Requirements and Goals

In Parts II and III, we examined solutions enabling the overlay and routing layer to play their fundamental role in reconfigurable content-based publish-subscribe. In this last part of the thesis, we make a further step and integrate the overlay and routing layers to form a complete system designed to work in large scale wired networks as well as in MANETs.

The evaluations presented in Parts II and III highlight the good results obtained by our approaches when considered in isolation. The protocols for the overlay layer are able to maintain a connected and acyclic topology in the presence of nodes that join and leave the network arbitrarily. Similarly, the reconfiguration protocols proposed for the routing layer are able to adapt routing information to topology changes with minimal communication cost, achieving a significant overhead reduction with respect to the strawman approach available in the literature.

These good results suggest that the combination of these layers can provide successful content-based information dissemination in large-scale dynamic network environments. The goal of this integrated evaluation is to confirm this expectation and to evaluate the interactions between the two layers. For example, LSTree maintains the overlay connected while minimizing the size of the area affected by reconfigurations. This aspect is likely to have a beneficial impact on our routing reconfiguration protocols by minimizing the number of hops travelled by subscriptions and unsubscriptions during reconfigurations. To an extreme, we can expect the improvements provided by this overlay to dwarf those provided by our optimized routing protocols. Our goal is to understand these interactions, enabling middleware designers to choose the simplest protocol among a those performing well in a given scenario.

In order to carry out an integrated analysis we developed a simulation framework comprising the protocols described so far. The framework is designed using a component-based architecture to enable the integration of different simulation modules depending on the specific protocols being simulated as well as on the spe-

cific simulation scenario. This allows us to integrate the various routing protocols and evaluate their relative performance incorporating different overlay modules on wired and wireless networks.

CHAPTER

FOURTEEN

Towards an Integrated Evaluation

Integrating the protocols for overlay management with those responsible for the maintenance of correct routing information is the last fundamental step in this research. The integration of the overlay and routing layer provides new insights on the operation of the protocols and serves as a guidance for the deployment of scalable middleware platforms in a real-world setting.

In this chapter we pose the basis for this last step. First we detail the choice of the most suitable overlay layers for deploying our middleware in peer-to-peer and mobile ad hoc networks. In particular, we provide a description of an overlay protocol for the latter scenario described in [67]. Then we consider the integration of the routing layer and discuss how our routing reconfiguration protocols can be combined with the selected overlay managers. Specifically, we extract a representative subset of routing reconfiguration protocols and discuss whether their requirements are satisfied by the overlay managers. Finally, we introduce the tool for our integrated evaluation: a simulation framework developed using OmNet++ [95].

14.1 Integrating the Overlay Layer

The first step in the deployment of a middleware solution is the choice of a suitable overlay layer for the target application scenario. In this section, we address this choice in the context of peer-to-peer and mobile ad hoc networks. In the first case, the choice is between the two protocols presented in Part III, while in the second it falls on WiTree, an overlay management protocol for mobile ad hoc networks developed by our research group.

14.1.1 Choice of the Wired Overlay

In the third part of this thesis we presented two overlay management protocols for large scale peer-to-peer networks: LSTree and DHTree. While both protocols can be applied in the design of reconfigurable publish-subscribe middleware, the LSTree protocol was designed with reconfigurable middleware as its primary application scenario. Its characteristics, and primarily its ability to localize topology changes to a small region of the overlay, make it particularly suited for the integration with the routing reconfiguration protocols presented in Part II.

As we already mentioned in Chapter 12, the overlay management techniques of the LSTree protocol integrate seamlessly with the routing reconfiguration of TIMED DEFERRED UNSUBSCRIPTION and can also be easily interfaced with the slightly more complex NOTIFIED DEFERRED UNSUBSCRIPTION and INFORMED LINK ACTIVATION protocols. The DHTree protocol, on the other hand, can be integrated only with TIMED DEFERRED UNSUBSCRIPTION, and it requires significant modifications to be used with the remaining protocols. In addition, the LSTree protocol was shown to be very efficient even in demanding reconfiguration scenarios. For these reasons, our choice for an integrated evaluation in large scale peer-to-peer scenarios falls solely on the LSTree protocol.

14.1.2 Choice of the Wireless Overlay

The choice of an overlay for wireless network falls inevitably outside the solutions presented in this thesis. Neither of the overlay protocols we presented is in fact designed to operate in a MANET environment, even though DHTree could in principle be deployed on top of the available DHT implementations for ad hoc wireless networks [5].

WiTree The problem of maintaining a tree overlay in ad hoc wireless networks has been successfully addressed by others in our research group [67]. The protocol they propose builds and manages a connected overlay in a MANET environment through an evolution and adaptation of the techniques used in the MAODV routing protocol [84].

As we mentioned in Chapter 10, MAODV maintains trees consisting of a set of multicast member nodes and a multicast group leader. The group leader sends periodic group hello messages (GRPH) to disseminate information about group membership to all group members as well as to detect and reconnect partitioned trees. This includes the information about the distance from the group leader used by the protocol to prevent cycles with a mechanism similar to that of our LSTree protocol.

When a link between two nodes fails as a result of mobility, the protocol designates the downstream node (the one farther from the leader) as the node responsible for reconnecting the tree by locating a new parent. Differently from our LSTree protocol, however, the nodes in MAODV cannot exploit node caches to locate new candidate parents as no underlying routing protocol is available. Therefore, they find their new parents using a route discovery procedure that consists of three phases. A node searching for a parent first request a new route

to the multicast tree. Then it collects a number of replies; and finally it selects one of these replies and reconnects to the tree.

Using three phases is necessary because requests are not unicast to one node at a time as in our LSTree protocol. Rather, a node that has lost the link to its parent broadcast a route request message (RREQ) to its direct neighbors. A node receiving this request can react in three different ways. If it is not a member of the multicast tree, it simply rebroadcasts the message and stores the identifier of the sending node to establish a path for the propagation of reply messages. If, on the other hand, it is a member of the multicast tree and its distance from the group leader is less than that of the requesting node, it replies by sending a route reply message (RREP) to the node from which it received the request. Finally, if neither of the above conditions holds, the receiving node simply discards the request without taking any action.

RREP messages follow the reverse path established by the propagation of the corresponding RREQ messages up to the requesting node. During this propagation, they establish a forward path to be used if the route is activated.

The requesting node waits for a specified timeout and collects all incoming replies. When this timeout elapses, it selects one of them and activates the corresponding route. To accomplish this, it sends a MACT, multicast activation message, along the forward path established by the selected reply.

A requesting node may fail to locate a suitable parent. In this case, it infers that the network has become partitioned and elects itself as a new group leader. The periodic GRPH messages broadcast by leaders allow nodes to detect such partitions and to take proper action to merge them again as soon as possible.

WiTree's improvements WiTree improves the behavior of the MAODV protocol with a set of extensions to optimize its use as an overlay network for publish-subscribe in mobile network environments. In the following, we provide a brief description of two of the most relevant extensions.

A first improvement of WiTree over MAODV regards the ability of the protocols to reconnect the tree without requiring the requesting node to be an endpoint of the newly located link. This increases the chances to reconnect in cases where MAODV would declare a partition even when connectivity is available. A second improvement allows the protocol to limit the cost of reconnection by shortening the path travelled by multicast activation messages. These two improvements allow WiTree to simplify the mechanism used by MAODV to reconnect partitioned subtrees by effectively using the same strategy used when repairing a link failure.

14.2 Integrating the Routing Layer

The routing reconfiguration protocols presented in Part II were designed on top of an abstract topology management system whose only task was to replace one communication link with another in a tree-based overlay network. This allowed us to design these protocols in a general way making minimal assumptions on the overlay management layer. The purpose of this section is to review these assumptions and to map them onto the characteristics of the overlay managers

we exploit in our integrated evaluation. The section is structured as follows. First we review the routing reconfiguration protocols described in Part II and select a subset of them for our integrated evaluation. Then we address their integration with the WiTree overlay manager and with LSTree.

14.2.1 Choice of the Routing Reconfiguration Protocols

The routing reconfiguration protocols described in Part II differ both in terms of performance and requirements on the overlay layer. Nevertheless the analysis we presented in Chapter 7 allows us to reduce the number of reconfiguration protocols for our integrated evaluation.

Specifically, simulation results showed that the performances of the two DEFERRED UNSUBSCRIPTION protocols are in most cases comparable and that differences are only visible in scenarios characterized by a very high event load. This drives our choice towards the TIMED DEFERRED UNSUBSCRIPTION protocol, as it combines a simpler implementation with looser requirements on the overlay management layer.

Similar reasoning holds for the choice between RECONFIGURATION PATH and INFORMED LINK ACTIVATION. Both protocols have the ability to confine the changes resulting from reconfiguration to the reconfiguration path and thus achieve similar performance improvements over STRAWMAN. However, INFORMED LINK ACTIVATION is applicable in a wider range of scenarios thanks to its ability to operate correctly in the presence of concurrent overlapping reconfigurations. Moreover its requirements on the overlay layer are looser than those of RECONFIGURATION PATH as the endpoints of a broken link only need to be notified about the identity of the endpoints of the new one and do not need to know the entire reconfiguration path.

These considerations determine the choice of the protocols we consider in our integrated evaluation: STRAWMAN, TIMED DEFERRED UNSUBSCRIPTION, and INFORMED LINK ACTIVATION.

14.2.2 Integrating Strawman and Timed Deferred Unsubscription

The discussion in Chapter 8 pointed out that TIMED DEFERRED UNSUBSCRIPTION is the protocol that can most easily integrated with the underlying overlay layer. Its interface with the overlay is in fact identical to the interface of the STRAWMAN protocol. Both only require the overlay to notify each dispatcher of the appearance or disappearance of links to neighboring dispatchers by means of the addLinkTo and removeLinkTo calls.

Both WiTree and LSTree provide these notifications as part of their normal behavior. In either protocol, a node detecting the disappearance of a link to a neighbors, either a parent or a child, makes a removeLinkTo call on its routing layer thus triggering the corresponding reconfiguration protocol. Similarly, an addLinkTo call is made by the protocol as soon as a connection request is accepted. In WiTree this happens when a MACT is generated or received by either of the endpoints of a new link. In LSTree, it happens when a PARENTREQUEST is

accepted by a candidate parent as well as when the corresponding acceptance message is received by the requesting node.

14.2.3 Integrating Informed Link Activation

The integration of the INFORMED LINK ACTIVATION protocol, on the other hand, is more complex. In this protocol, each of the endpoints of a removed link must be notified of the identity of the corresponding endpoint of the replacement link. This gives the protocol a symmetric structure in which each of the endpoints of the old link sends its own ACTIVATE message.

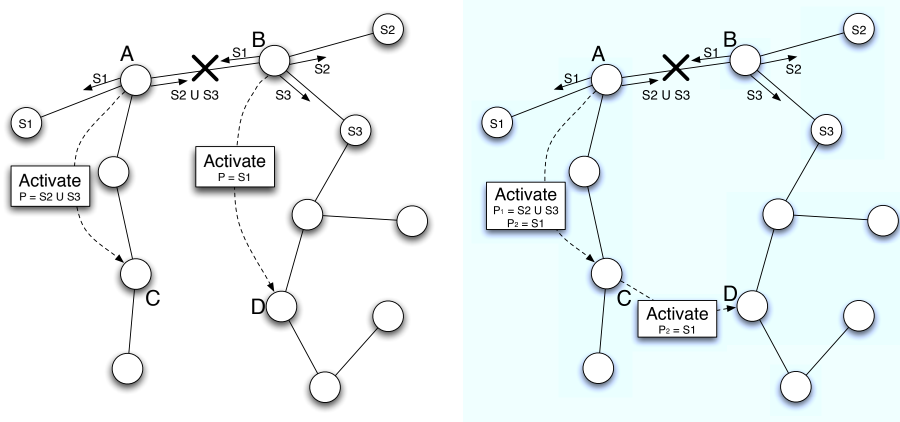
However, both WiTree and LSTree operate in an asymmetric fashion. In both cases only the endpoint(s) that are farther from the root node take an active part in the reconfiguration process and, in general, only these endpoints are notified of which new link has been chosen. This seems to make the integration of these protocols with INFORMED LINK ACTIVATION particularly complex; however, a quick analysis of the INFORMED LINK ACTIVATION protocol reveals that the two ACTIVATE can be replaced by a single message sent by only one of the endpoints of the removed link.

In the following we describe how this leads to a simple integration of INFORMED LINK ACTIVATION with both of the considered overlay managers. For clarity, we first address the integration with WiTree, and then define the additional details required for the integration with LSTree.

Deployment of Informed Link Activation over WiTree Let us consider the situation in Figure 14.1(a) in which the link between dispatchers A and B is replaced by a new one between C and D. According to the description of Section 6.2, dispatcher A sends send an ACTIVATE to dispatcher C containing all the patterns in its subscription tables to which only B was subscribed, $P = S2 \cup S3$. Similarly, dispatcher B informs D of the patterns to which only A was subscribed with another ACTIVATE containing $P = S1$.

However, the content of either ACTIVATE can easily be computed by dispatcher A using only the information stored in its own subscription table. Let us consider the patterns contained in B's ACTIVATE, $P = S1$ in the figure. These correspond to patterns that were used only to route events across the removed link to dispatcher A and possibly beyond it. In other words, they correspond either to subscriptions issued by A itself or to subscriptions used by A to route events to one of its neighboring dispatchers other than B. A can therefore easily compute the contents of B's ACTIVATE by considering the patterns in its table that to which B is not subscribed.

This reasoning leads to an *asymmetric* version of the INFORMED LINK ACTIVATION protocol whose operation is summarized in Figure 14.1(b). One of the two endpoints of the broken link, the one responsible for controlling the reconnection process, A in this case, computes both sets of unnecessary subscriptions, P_1 and P_2. P_1 contains the patterns used by A only to route events across the removed link, while P_2 contains all the subscriptions in A's subscription table that are *not* associated to the removed link. It's important to observe that these two sets do not represent a partition of A's subscription table, although this is the case in



(a) INFORMED LINK ACTIVATION with two ACTIVATE messages

(b) INFORMED LINK ACTIVATION with a single ACTIVATE message

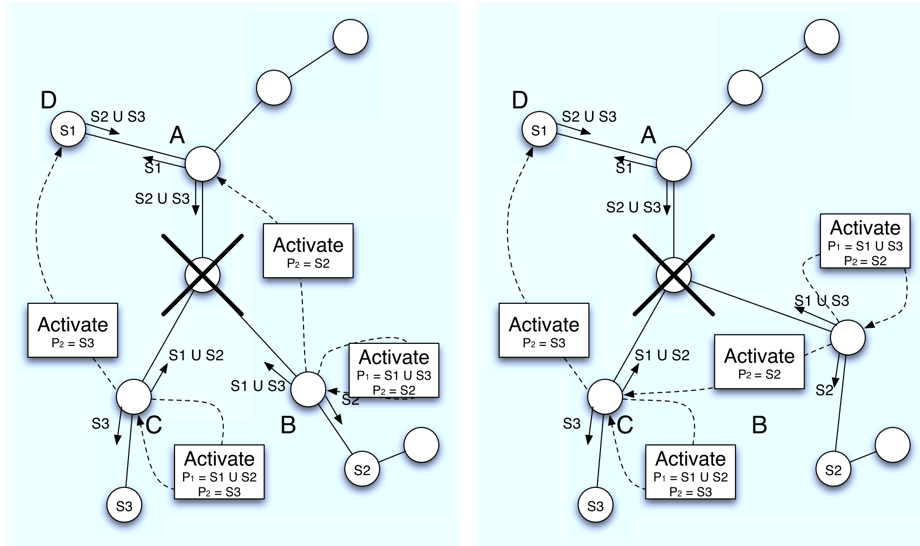
Figure 14.1: Integrating the INFORMED LINK ACTIVATION protocol with the overlay manager. The figure shows that the two ACTIVATE's used by the protocol can be substituted by a single message containing all necessary information. This makes the protocol asymmetric and simplifies its integration with the WiTree overlay manager. For clarity the figure shows only the subscription tables associated to the nodes involved in the reconfiguration. If a node has recorded two subscriptions $S1$ and $S2$, they are shown in the figure as $S1 \cup S2$.

the figure. If the removed link part of the pattern tree for some subscription s, then A subscription table will contain at least two entries for s: one associated to B and the other associated to some other dispatcher, possibly A itself. This subscription is not included in either of the two sets P_1 or P_2.

Once A has computed P_1 and P_2 it sends them in a single ACTIVATE to the endpoint of the new link it's own subtree, C in the figure. This dispatcher uses P_1 for its reconfiguration and propagates the ACTIVATE to D, this propagated message can contain only P_2. Dispatcher D takes the content of P_2 and uses it in its reconfiguration process exactly as it would have done with the set of pattern in B's ACTIVATE.

The only remaining steps in the INFORMED LINK ACTIVATION protocol are the two timeouts and flush messages. These are handled exactly as in the original version. An unsubscription timeout is triggered at each of the endpoints of the removed link, while a subscription timeout is triggered at the endpoints of the new link. In the evaluation presented in the following chapters we consider an implementation of INFORMED LINK ACTIVATION without flush messages. Nevertheless, flush messages can be propagated by the endpoint of new links exactly as in the symmetric case.

It is worth observing that this asymmetric version of INFORMED LINK ACTIVATION does *not* define a new protocol. Rather it simply provides a different way to implement the same underlying mechanism, providing for a simpler deployment



(a) Scenario with two ancestor connections.

(b) Scenario with one ancestor and one sibling connection.

Figure 14.2: Integrating the INFORMED LINK ACTIVATION protocol with the LSTree overlay manager. The figure shows that one of the two ACTIVATE's is always "sent" by a child of a failed node to itself. The other is instead propagated to the new parent and contains the set of patterns which were *not* on the link from the child to the failed node.

over the WiTree overlay manager.

Deployment of Informed Link Activation over LSTree The integration of the protocol is analogous in the case of LSTree. At first sight, this case might appear more complex because multiple links are being replaced and because one of the endpoints of these links disconnects from the overlay. However, a quick analysis shows that we can apply the same reasoning we applied in the case of WiTree.

Let us consider the situation in Figure 14.2(a). The disconnection of a node leads to the disappearance of the 3 links to nodes A, B, and C, while the overlay manager re-establishes connectivity by adding 2 new links: from C to D, and from B to A. Because a node left the overlay only 2 of the 3 links are replaced, and the other is simply lost. Thus, to integrate the INFORMED LINK ACTIVATION protocol, it is sufficient to select two of the removed links and associate each of them with one of the new ones.

In principle, any pairing of new and old links can be used. However, the use of the LSTree protocol naturally suggests one such pairing. The link from the failed node to its parent is the one that is lost. Each of the links to its children is instead associated to the new link found by the corresponding child using the LSTree protocol. With reference to Figure 14.2(a), the link from the failed node

to C is replaced by the new $C - D$ link, while the link to B is replaced by $B - A$.

Once new links have been associated to removed ones, reconfiguration can be performed using the asymmetric version of the INFORMED LINK ACTIVATION protocol. Each of the children of the failed node computes the sets P_1 and P_2 as discussed above. However, the propagation of the ACTIVATE messages is simplified with respect to the WiTree case. The first hop of the ACTIVATE is, in fact, not necessary as the message would be sent from each of the failed node's children to itself. Consequently, each of the failed node's children only forwards an ACTIVATE message containing the set P_2 to the other endpoint of the new link. It then uses the set P_1 locally to avoid propagating unnecessary unsubscriptions to the other endpoint.

With reference to the figure, the set P_1 allows node C to prevent the propagation of the subscriptions $S1$ and $S2$ to dispatcher D, while the corresponding set P_2, allows D to prevent the propagation of $S3$ to dispatcher C. As a final note, we observe that timeouts and flush messages are managed as they would in the normal operation of the INFORMED LINK ACTIVATION protocol except that no timeout is clearly triggered at the failed node.

The outlined reasoning allows the INFORMED LINK ACTIVATION protocol to be integrated with LSTree without modifying the overlay. However, it is worth pointing out that the association between new and old links *suggested* by the LSTree protocol is not always perfect. Figure 14.2(b) shows a situation in which the link to dispatcher B is replaced by a new one from B to C. In this case, the set P_2 contained in the ACTIVATE sent from C to D allows D to propagate an unnecessary subscription along the new link. Subscription $S2$ is not contained in P_2, and is therefore unnecessarily propagated to C and then removed at the expiration of the timeout.

This situation arises when the LSTree protocol recovers a failure starting from the Sibling Set. However, as we pointed out in Section 10.4, candidate parents are often chosen among the ancestors of failed nodes or among their (downstream) neighbors. As a result, the propagation of unnecessary subscription generally remains limited. Nevertheless we are currently investigating smarter ways to associate new and old links thus increasing the performance of INFORMED LINK ACTIVATION over the LSTree protocol.

14.3 Integrated Simulation Framework

A significant part of our research efforts was devoted to a careful simulation of the behavior of our protocols, both to evaluate their individual performance and to understand their interaction when applied together. The product of our work, however, is not limited to the performance results we obtained but it also includes a simulation framework that can be exploited to evaluate the performance of our middleware in a broad range of scenarios. Middleware designers can exploit this framework to evaluate the performance of protocols for the overlay, routing and also for the event-recovery layer, integrating the work presented in this thesis. Similarly, application developers can evaluate the performance of applications built on top of our middleware without deploying them into a real large-scale test-bed.

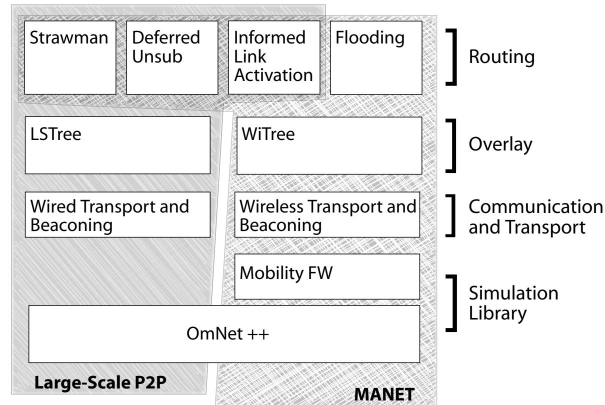


Figure 14.3: Architecture of the integrated simulation environment.

The basis for our simulation framework is OmNet++ [95], an open-source discrete-event simulation environment, successfully used by several researcher to model communication networks, complex IT systems, queueing networks and hardware architectures. An OmNet++ simulation consists of set of interacting components called *modules*. These components can be programmed in C++ and assembled together into larger components to facilitate the reuse of previously programmed modules in new simulation studies.

According to OmNet++'s philosophy our simulation framework also exhibits a component-based architecture. Each of the protocols presented in the thesis is implemented as an independent module or as a combination of modules. In both cases, protocol modules specialize abstract module components. This allows us to define two generic components representing a wired or a wireless mobile host.

Figure 14.3 depicts a high-level architectural view of the simulation framework. The lowest level consists of the OmNet++ core library and the Mobility Framework used in the simulations of mobile ad hoc networks. The three layers on its top, instead, constitute our integrated simulation architecture. The small boxes within each layer represent the protocols implemented in the simulator, while the big vertical boxes spanning all the layers show the components used in the wired and wireless versions of the simulator. In the following, we provide a brief description of the layers in this architecture, pointing out which of the modules are used in wired and/or in wireless simulations.

Transport and Network Layer The lowest level of our simulation architecture enables communication between the hosts in our distributed application. The simulator includes two version of this communication layer: one for wired and one for wireless networks. The version for wired networks consists of a simple communication module that simulates transmission and propagation delays between the hosts in the network. The version for wireless networks, on the other hand, is based on Omnet++ Mobility Framework.

In both cases, the communication layer implements a reliable communication primitive. In the wired case, all messages are exchanged using reliable TCP channels, while in wireless networks communication is achieved with a mixture of reliable and unreliable communication. Subscriptions, unsubscriptions and some of the overlay messages are propagated using reliable one-hop communication,[1] while events are propagated with unreliable one-hop broadcast.

The Overlay Layer The overlay layer is responsible for maintaining connectivity among the components of the distributed event dispatcher. Each of the modules in this layer implements a specific topology maintenance protocol. In the current version, the simulator incorporates an implementation of LSTree, as well as an implementation of WiTree, the topology maintenance protocol for Mobile Ad Hoc wireless Networks presented in [67] and described in Section 14.1.2.

The routing layer The routing layer consists of an abstract routing component, the *Event Dispatcher*, specialized according to the routing and reconfiguration strategy adopted in the simulation. The use of this abstract component representing a node of a generic distributed event dispatcher enables the integration of different routing strategies without recompiling the simulation code.

In its current version, the framework includes two direct specializations of this component: a subscription forwarding dispatcher that can be used both in a wired and in a wireless setting and a broadcast-based structure-less event-flooding dispatcher that serves as a baseline for comparison in the wireless setting. The subscription forwarding dispatcher is further specialized in three reconfigurable solutions corresponding to the three reconfiguration protocols we consider in this and the following chapters: STRAWMAN, TIMED DEFERRED UNSUBSCRIPTION, and INFORMED LINK ACTIVATION.

Observer Modules Alongside the protocol stack described above, the simulator incorporates a set of modules responsible for recording relevant data during the simulation. These monitoring components provide the basic mechanism to evaluate the performance of our protocols by measuring overhead, event delivery, latency and so on. In addition, they constitute a useful tool during the development of new protocol modules. Possible uses include checking the system for inconsistencies, performance bottlenecks, and other potential problems.

[1] We assume no underlying routing protocol in the wireless case.

CHAPTER

FIFTEEN

Evaluation in P2P Scenarios

In this chapter we present the first part of our integrated evaluation carried out using the framework described in Section 14.3. We consider a large scale peer-to-peer scenario with nodes that join and leave the network at arbitrary times and assess the performance of our routing reconfiguration and overlay maintenance protocols. The chapter is structured as follows. Section 15.1 introduces our simulation setting and motivates our choices for the simulation scenarios. Section 15.2 presents the results we obtained, and Section 15.3 concludes the chapter with some concluding remarks.

15.1 Simulation Setting

The scenarios used in the simulations presented in this chapter are obtained from a combination of the scenarios we considered in Chapter 7 for our routing reconfiguration protocols, with the Gnutella scenario [53] we used in Section 10.4 to evaluate our LSTree protocol. As a result, the configuration of the LSTree protocol is the same as that used in Chapter 10.4, while the configuration of our routing protocols has been adapted to match the time required by the LSTree protocol to reconnect a broken overlay.

In the following we first report a summary of the most relevant overlay parameters, and then presents the parameters we used for our overlay protocols. Finally, we describe the set of scenarios we considered in our simulation study.

Overlay Configuration The overlay maintenance protocol is configured to maintain a connected overlay topology characterized by a maximum node degree of 5 and without any limit on the minimum node degree (RDGM instance). The length of the upstream chain is set to 3, while the probability of choosing an ancestor versus a sibling from the Regional cache is set to 0.5. The size of the Global Cache is set to a maximum of 10 references to nodes, while the number of references for the Downstream cache in refusal messages is set to 3. Periodic updates to the global cache are done every 250s, with nodes propagating the

contents of their cache to 2 neighbors at random. Nodes receiving these cache updates re-propagate them with a probability of 0.3.

Routing Configuration According to what we mentioned in Chapter 14 both our optimized protocols are configured to operate without flush messages. This means that they rely only on timeouts to trigger the propagation of unsubscriptions as well as of tagged subscriptions in the case of INFORMED LINK ACTIVATION. Clearly, the values of these timeouts must be configured according to time required by the overlay protocol to reconnect the tree after a failure.

In Chapter 7, reconnection is always achieved within $0.1s$ and both timeouts are configured to the value of $.15s$. For the simulations in this chapter, we measured that the LSTree protocol is able to reconnect the overlay within $1s$ in 85% of the cases; as a result, we opted for timeout values of $1.5s$ both for the Subscription and the Unsubscription timers.

Simulation Scenarios As previously mentioned, the simulation scenario consist of a combination of the Gnutella scenario of Section 10.4, with an adaptation of the routing scenarios used in Chapter 7. In the following we describe the characteristics of this aggregate scenario and motivate the most relevant choices.

The peer-to-peer network is built by extracting a random subset of nodes from the trace files provided by the authors of [53]. This allows us to vary the scale of the system from a minimum of 100 to a maximum of 2700 nodes simultaneously awake in the system. As a reference point when varying the other parameters, we consider networks with approximately 270 nodes corresponding to 10% of the nodes in the Gnutella trace.

Unless otherwise specified 20% of the nodes in the system express interest in receiving events. To maintain a fairly constant number of subscribers despite nodes joining and leaving, each node is allowed to subscribe with a probability corresponding to the desired percentage of subscribers. Each subscriber subscribes exactly to 7 patterns chosen from a set of 400. This allows us to simulate large-scale systems without saturating the set of patterns too soon.

All the nodes in the system are allowed to publish events with frequencies ranging from a 15 to 3000 published events per minute in the whole system. Each events is configured to match a maximum of 40 patterns. This, combined with a subscriber density of 20%, results an average of 10% of the nodes being interested in each event. Finally each measured simulation interval consists of 7200 simulated seconds.

15.2 Simulation Results

We present our simulation results in three steps. First we analyze the performance of our routing reconfiguration protocols on top of the LSTree overlay. Second, we relate their performance improvements to the cost of overlay maintenance. Finally, we relate these improvements with event notifications, comparing the advantages obtained by the optimized protocols with the fixed cost for the delivery of events to interested subscribers.

15.2.1 Routing Reconfiguration

The first step in our integrated evaluation is the analysis of the performance of our routing protocols. While this may seem a repetition of the analysis presented in Chapter 7, our evaluation highlights significant differences from the data gathered in such setting. The results in Chapter 7 are obtained using random topology changes that replace a link between two nodes with a new link between two other nodes chosen at random. The results in this section, on the other hand, consider the behavior of the reconfiguration protocols in response to the topology changes induced by the LSTree overlay management mechanisms.

The relevance of these differences is twofold. First, they confirm that the performance of our protocols is dependent on the specific overlay maintenance strategies. Second, they validate the efficacy of LSTree and show its ability to enhance the performance of our reconfiguration protocols.

Our analysis is presented in three steps. First we evaluate the protocols' performance with a variable system scale. Second we assess the impact of the density of subscribers and finally we observe how the protocols are affected by different rates of publications.

Throughout this Section 15.2.1, the results shown in the plots include subscriptions unsubscriptions and control messages exchanged by routing reconfiguration protocols. However they do not include the control messages exchanged at the overlay layer as these are examined later, in Section 15.2.2.

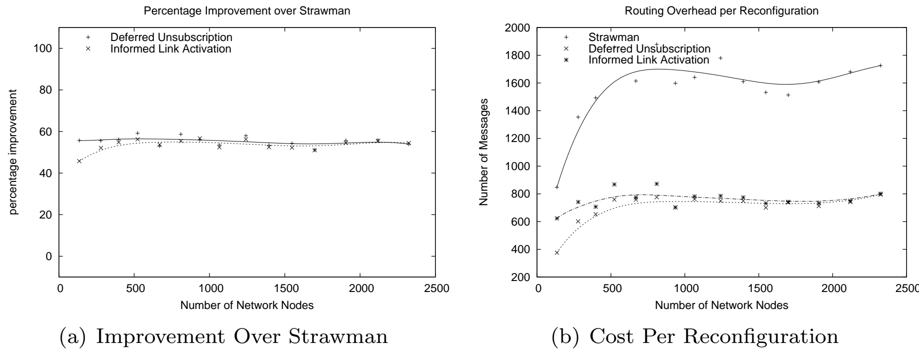
Performance with Variable System Scale

Figure 15.1 shows the results obtained when the scale of the system varies between 100 and 2300 nodes. The left plot shows the percentage improvement obtained by Timed Deferred Unsubscription and Informed Link Activation over Strawman, while the right plot shows the unit cost of each reconfiguration expressed as the number of messages exchanged to reconfigure routing information.

The first observation we can make by examining the plot in Figure 15.1(a) is the difference in the relative performance of the two optimized protocols with respect to the data presented in Chapter 7. In our evaluation of routing protocols, Informed Link Activation achieved a 70% performance improvement over Strawman in a scenario with sparse subscribers and a low event load corresponding to the one in Figure 15.1(a), while the performance of Timed Deferred Unsubscription was lower, with only a 50% improvement in the same scenario.

The plot in Figure 15.1(a), however, shows a very different behavior. The performance of the Timed Deferred Unsubscription protocol is higher with a 60% improvement over Strawman, but most notably the Informed Link Activation protocol is almost unable to go over the performance of Timed Deferred Unsubscription. The reason for this seemingly weird behavior lies in the combination of the two protocols with our overlay manager.

The improvements obtained by Informed Link Activation over Timed Deferred Unsubscription in Chapter 7 derive from its ability to avoid the propagation of unnecessary subscriptions by recognizing the patterns used only to route events over removed links. This mechanism is a generalization of the optimization carried out by all the optimized protocols, including Timed Deferred



(a) Improvement Over Strawman

(b) Cost Per Reconfiguration

Figure 15.1: Improvement over Strawman and cost per reconfiguration as system size increases.

UNSUBSCRIPTION, whenever a dispatcher is an endpoint of both a new link and the corresponding vanished link. In such cases, all the optimized protocols behave in the same manner and avoid propagating the subscriptions directed towards a broken link.

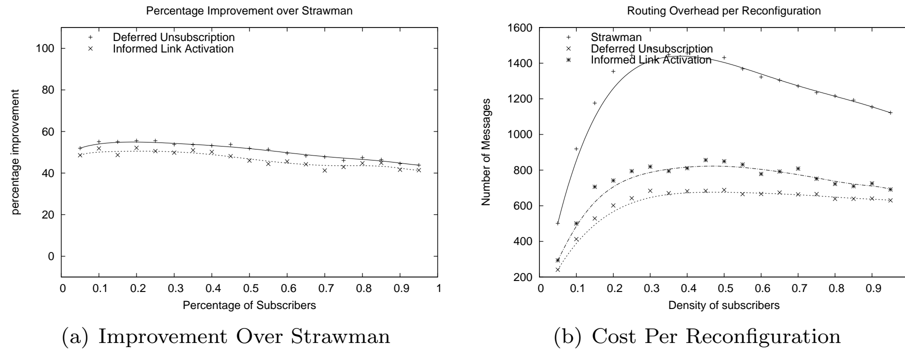
The optimization applied by INFORMED LINK ACTIVATION is therefore only relevant in cases where the endpoints of new and old links differ, a rare situation when the LSTree protocol is used. This protocol, in fact, explicitly chooses the new links to repair a failure with the objective of minimizing reconfiguration cost.

Specifically, LSTree always tries to repair failures regionally by connecting the downstream neighbors of a failed node to the failed node's ancestors or to each other. This means that the first endpoint of a newly added link always coincides with one of the endpoints of a vanished link. Moreover, even the second endpoint of a new link is likely to be a neighbor of the failed node and thus an endpoint of a corresponding vanished link. The net effect, therefore, is that TIMED DEFERRED UNSUBSCRIPTION is able to apply its "shared dispatcher" optimization in the vast majority of the cases, operating exactly as the INFORMED LINK ACTIVATION protocol in terms of propagated subscriptions and unsubscriptions.

When connections are made to nodes that are not among the failed node's neighbors, the INFORMED LINK ACTIVATION protocol is able to carry out its optimization and to reduce the number of unnecessary subscriptions. However, it does so by propagating an ACTIVATE message containing a potentially very large number of event patterns. In the scenarios of Chapter 7, this activation message prevents the propagation of a large number of subscriptions for a large number of hops. In the present setting, however, the resulting improvement is insufficient to outweigh the cost of the ACTIVATE message.

The reason is that the LSTree protocol is able to minimize the length of the reconfiguration path[1] in a vast majority of cases. As a result the unnecessary subscriptions of TIMED DEFERRED UNSUBSCRIPTION propagate only for a shorter distance, making the improvements achieved by INFORMED LINK ACTIVATION

[1] More properly, the size of the reconfiguration area.



(a) Improvement Over Strawman

(b) Cost Per Reconfiguration

Figure 15.2: Improvement over Strawman and cost per reconfiguration as density of subscribers increases.

almost negligible.

The net effect of these interactions is an increase in the overall performance of Timed Deferred Unsubscription and a corresponding decrease in the performance of Informed Link Activation due to the higher relative cost of its activate messages. These results confirm that we achieved our goals in the design of the LSTree protocol, improving the overall efficiency of the reconfiguration process.

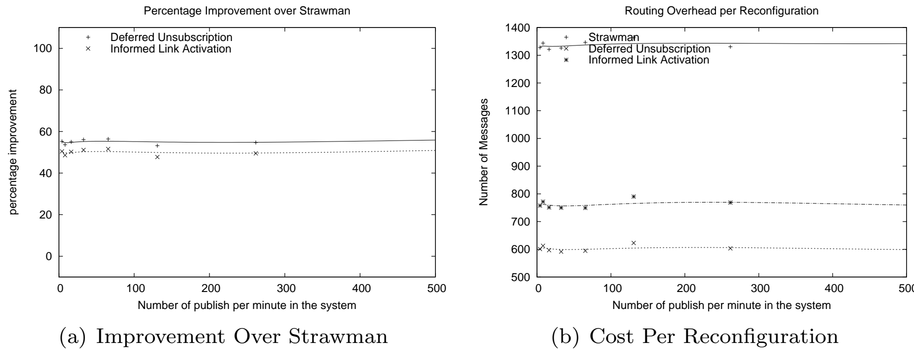
Effect of Subscriber Density

The second aspect we consider in our integrated evaluation is the effect of the density of subscribers on the relative performance of our routing reconfiguration protocols. In Figure 15.2, we present the results obtained with a fixed network size of 270 nodes and a variable density of subscribers ranging from 5% to 100%.

The data depicted in Figure 15.2(a) confirms the behavior already outlined in the observations we made in Chapter 7. The improvement achieved by our optimized protocols over Strawman is greatest when subscribers are sparse because subscriptions and unsubscriptions propagate for longer distances on the tree. The optimized protocols manage to limit this propagation by anticipating the subscription phase, while Strawman incurs in a very high cost associated to the propagation of unsubscriptions followed by subscriptions to very large portions of the tree.

In addition, the plot in Figure 15.2(b) shows another very interesting aspect. The unit cost of reconfiguration has a maximum when the density of subscribers is around 40%. When the density of subscribers is low, the total number of patterns in the system increases with the density of subscribers. The maximum is reached when the tree starts becoming dense with subscriptions for the same pattern. When this happens, adding a new subscriber has the effect of reducing the propagation of subscription messages as pointed out in the observations in Section 5.2.

Interestingly, the concavity of the unit cost of reconfiguration is most evident



(a) Improvement Over Strawman (b) Cost Per Reconfiguration

Figure 15.3: Improvement over Strawman and cost per reconfiguration as the frequency of publications increases.

in the STRAWMAN protocol. This is because the optimized solutions already benefit from the increased density of subscribers resulting from the propagation of subscriptions before to unsubscriptions.

Impact of Publish Rate

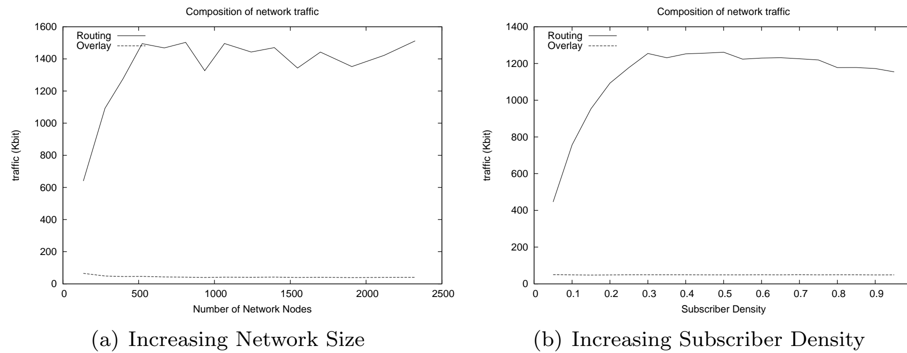
The next parameter we consider is the frequency of event publications and its effect on the reconfiguration of routing. In Chapter 7, we observed that the TIMED DEFERRED UNSUBSCRIPTION protocol was negatively affected by very high event loads due to the impact of misrouted events.

In this section we set the size of the system to approximately 270 nodes, the density of subscribers to 20%, and we vary the event load from a minimum of 15 publish operations per minute in the overall system to a maximum of 500. These rates are obtained with all the nodes in the system acting as publishers; nevertheless the same values can be obtained with fewer publishers each with a higher publish frequency. It is worth observing that while these values are lower than the extreme ones considered in Chapter 7, they represent realistic situations in peer-to-peer environments.

The plots in Figures 15.3(a) and 15.3(b) show that at the considered publish rates, the cost of reconfiguration is almost unaffected by the frequency of publications. The cost for forwarding misrouted events is much lower than the cost for forwarding subscriptions, unsubscriptions and control messages. For example, with 500 publish operations per minute in the system, TIMED DEFERRED UNSUBSCRIPTION propagates less than 5 misrouted events per reconfiguration. This is primarily due to the event load considered in Figure 15.3, but is also a consequence of the short reconfiguration path determined by the LSTree protocol.

15.2.2 Routing Reconfiguration Versus Overlay Maintenance

Up to this point we have concentrated on the performance of our routing reconfiguration protocols to understand their behavior when deployed on top of LSTree.



(a) Increasing Network Size (b) Increasing Subscriber Density

Figure 15.4: Routing versus overlay cost with increasing network size and subscriber density.

However, the traffic generated by these protocols is not the only relevant component in determining the cost of reconfiguration. The LSTree protocol itself must exchange messages to maintain a connected overlay in the presence of failures.

In this section, we address this issue and relate the cost of reconfiguring routing information with the cost associated to the LSTree protocol. In this comparison, however, we cannot directly relate the number of messages exchanged by the two protocols. The messages exchanged by LSTree are in fact very diverse and can range from empty ICMP packets to digests containing information on possibly large sets of nodes.

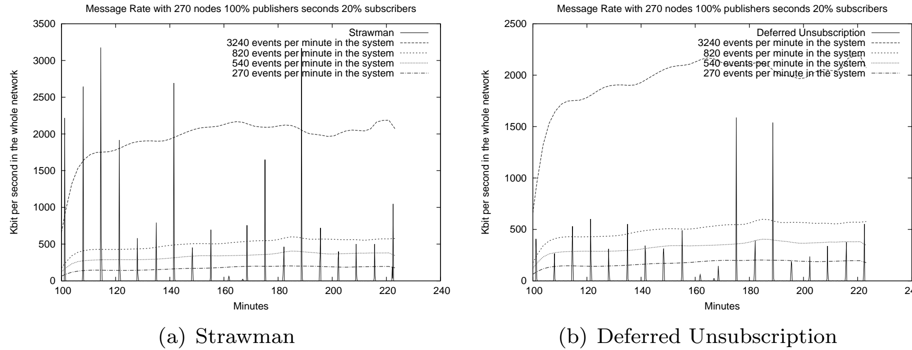
As a result, in the following, we assign a cost to each message according to the size of the data it contains. For example, a message containing four node identifiers is assumed to be 16 bytes large plus the size of TCP headers, assuming that each node identifier consist of an IP address.

While this evaluation is easy for overlay messages, the size of the messages exchanged by the routing protocols is dependent on the specific application using the middleware. In the following, we adopt a size of 30 bytes plus headers for events, subscriptions and unsubscriptions.

Figure 15.4 depicts the composition of overhead in two of the situations considered in the previous section: variable scale, and variable density of subscribers; variable publish rate is not shown due to the negligible impact of misrouted events. In all cases, the cost for the maintenance of routing information is a lot larger than the cost associated to overlay maintenance. On the one hand, this confirms the efficiency of our LSTree protocol; on the other hand, it motivates the optimizations carried out by the routing reconfiguration layer.

15.2.3 Impact of Optimized Protocols

The data we just presented confirms the results described in the first two parts of this thesis. Both the optimized routing reconfiguration protocols and the LSTree overlay maintenance layer effectively manage to reduce the overall traffic generated by reconfigurations by a significant amount. In this section, we compare



(a) Strawman (b) Deferred Unsubscription

Figure 15.5: Reconfiguration cost versus event traffic in a simulation run.

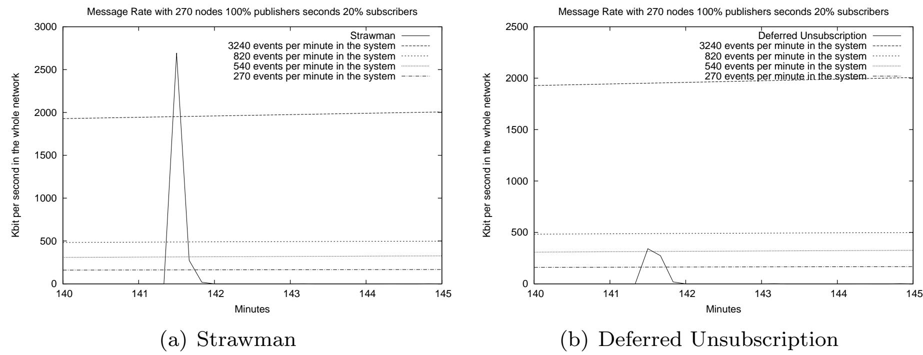
the improvements achieved by our protocols with the total cost of disseminating events.

The first observation we can make is that our protocols will most likely have a greater impact in situations with very restrictive event patterns. In such cases, event traffic is reduced to a minimum and the reconfiguration of routing information, if not carried out properly, may significantly increase the cost of the system. Our optimizations, however, are also relevant in scenarios characterized by a higher event traffic even when the total number of event notifications is much larger than the total number of subscriptions and unsubscriptions.

To assess this relevance, we must compare the cost of routing reconfigurations and the cost of event delivery in a meaningful way. If we were to consider the total number of messages exchanged in an arbitrarily long period of time, we would wrongly conclude that the impact of reconfiguration over the traffic generated by the system is minimal. One could argue that the cost of event routing is generally larger than the cost of a single reconfiguration, also by virtue of the short reconfiguration path determined by the LSTree protocol. However, the traffic associated to events is generally a constant component that is always present while the system is operating. Reconfiguration, on the other hand, occurs in bursts and generates traffic only when the topology changes as a result of a node joining or leaving the system. As a result, the total number of messages in a given time period provides a misleading measure.

A better way to measure the impact of our protocols is to evaluate how traffic varies with time. In Figure 15.5 we compare the traffic associated to reconfiguration, i.e. routing plus overlay, with the traffic associated to the dissemination of event notifications at several publish rates in a network of 270 nodes with 10% receivers per event. Clearly the cost of event dissemination increases as the frequency of publications increases. Nevertheless, the spikes associated to the Strawman solution correspond an even higher traffic, albeit for a limited period of time, with publish rates up to 3200 events per minute in the system.

Figure 15.6 shows a closeup view of the same situation. The plots clearly show that our optimized solutions can significantly improve the performance of



(a) Strawman

(b) Deferred Unsubscription

Figure 15.6: Reconfiguration cost versus event traffic in a 5 minute period.

the middleware in large-scale peer-to-peer scenarios. In a peer-to-peer setting, middleware nodes are likely to be deployed on home users' computers equipped with large but not unbounded bandwidth. A solution like Timed Deferred Unsubscription, allows these systems to achieve reconfiguration with only a minimal impact on the overall traffic, thereby enhancing the overall performance of the middleware.

15.3 Concluding Remarks

In this chapter we presented an integrated evaluation of our LSTree and routing reconfiguration protocols. Results confirm the ability of the LSTree protocol to reduce the cost of reconfiguration by localizing topology changes to a very small region of the overlay. This allows the TIMED DEFERRED UNSUBSCRIPTION protocol to improve its performance to the point of being preferable to the more complex INFORMED LINK ACTIVATION solution.

Overall, the two optimized protocols achieve significant improvements over STRAWMAN by reducing the traffic associated to the reconfiguration. In the near future, we plan to confirm these promising results with a real test-bed implementation to be deployed on standard personal computers and even on bandwidth-constrained devices in a peer-to-peer setting.

CHAPTER
SIXTEEN

Evaluation in MANETs

In this chapter we present the second of our integrated evaluations. We consider the deployment of content-based publish-subscribe middleware in a MANET environment by combining our routing reconfiguration protocols with the WiTree overlay manager.

The characteristics of mobile ad hoc networks represent a challenge for the development of effective communication middleware. The mobility of hosts and the unreliability of the wireless medium clash with the assumptions made in traditional middleware development. For this reason, most solutions for communication in MANET's depart from a traditional routing approaches and propose a communication based on epidemic algorithms or other unstructured routing strategies.

The protocols presented in this thesis, however, enable an efficient reconfiguration of routing information even in networks characterized by frequent topology changes. This motivates our study of mobile wireless networks as a scenario that can greatly benefit from the solutions presented in this thesis.

The goal of the evaluation presented in this chapter is twofold. On the one hand we aim to validate our optimized routing reconfiguration protocols in a further scenario. On the other hand, we aim to assess the effectiveness of tree based content-based routing in the context of wireless network.

The chapter is structured as follows. Section 16.1 introduces our simulation setting and motivates our choices for the simulation scenarios. Section 16.2 presents the results we obtained, and Section 16.3 concludes the chapter with some concluding remarks.

16.1 Simulation Setting

Our reference simulation scenario consists of a network of mobile nodes arranged in a square area of 1 square kilometer. We assume a *free space* radio propagation model. Nodes have a communication range of $200m$ obtained from standard values for carrier sense and receiving thresholds. They move according to a random

waypoint mobility model [60], with a pause time of 5 seconds and a speed of $1m/s$ unless otherwise specified.

The wireless network consists of 50 nodes, all acting as event publishers. In addition, 20% of the nodes issue subscriptions to 7 event patterns chosen from a set of 96 available patterns. Each generated event matches a maximum of 9 patterns yielding an average number of receivers per event corresponding to 10% of the nodes in the network.

In our tests we vary this scenario by individually changing each of the above parameters. First we vary the number of nodes from 10 to 100; then we vary the percentage of subscribers from 5% (2% of nodes receiving each event) to 100% (50% of nodes receiving each event); then we vary node speed from 0.2 to $2m/s$; and finally we vary the frequency of publications from a minimum of 1 to a maximum of 300 publish operations per second in the entire system. Each simulation run consists of 400 simulated seconds comprising $200s$ of setup time and a $200s$ measured interval.

Our reconfiguration protocols are configured according to the characteristics of the WiTree overlay manager. Specifically, WiTree requires a minimum time of $1s$ to reconnect the overlay and generally requires less than 1.5 seconds: consequently, we set both the Subscription and Unsubscription timeouts to $2s$ for both reconfiguration protocols.

16.2 Simulation Results

Similarly to what we did in Chapter 15, we present our simulation results in three steps. First we analyze the performance of our routing reconfiguration protocols on top of the WiTree overlay. Second, we relate their performance improvements to the cost of overlay maintenance. Finally, we address the performance of our approach as a whole and compare it to the performance of Event Flooding.

16.2.1 Routing Reconfiguration

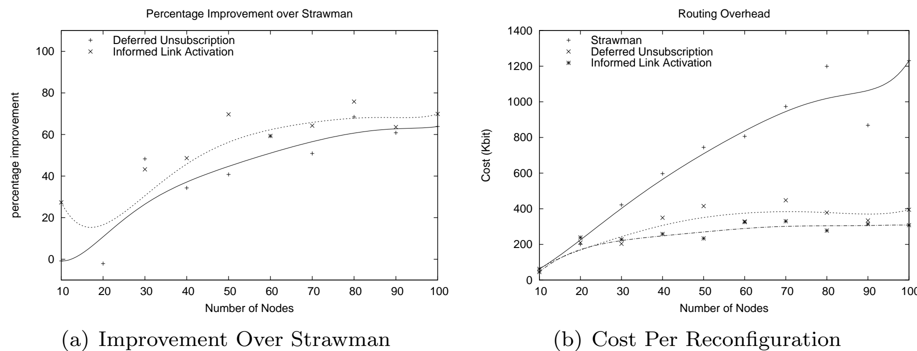
The first step in our integrated evaluation is the analysis of the performance of our routing protocols. Our results are presented in two steps. First we address the performance of our protocols with a variable number of network nodes and then we analyze their behavior with variable node speed.

Varying the Number of Nodes

Our first set of measurements examines the behavior of our routing reconfiguration protocols as the number of nodes in the simulation area varies from a minimum of 10 nodes to a maximum of 100.

The results depicted in Figure 16.1 show that differently from the wired case (Chapter 15), the improvements of Informed Link Activation play a more significant role and the protocol is able to outperform Timed Deferred Unsubscription although by a less significant amount than in the results of Chapter 7.

The reason for this behavior lies in the approach to reconfiguration adopted by WiTree, which causes it to repair the tree with a longer reconfiguration path



(a) Improvement Over Strawman (b) Cost Per Reconfiguration

Figure 16.1: Improvement over Strawman and cost per reconfiguration as the number of nodes in the system increases.

than the LSTree protocol used in the wired setting. Moreover in WiTree, the endpoints of old and new links coincide much more rarely than in the case of the LSTree protocol. This prevents Timed Deferred Unsubscription from applying its "shared dispatcher" optimization and increases the importance of the improvements provided by the Informed Link Activation solution.

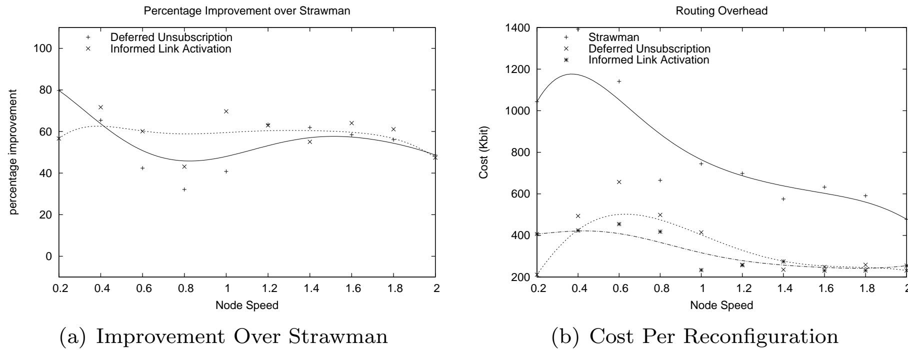
The plot in Figure 16.1(a) shows that both our optimized protocols perform consistently better than Strawman, except in the case of very small networks where only Informed Link Activation is able to perform better. In addition, the improvement of both protocols increases with the number of nodes in the network because the unnecessary subscriptions and unsubscriptions propagated by Strawman can travel for longer distances.

Figure 16.1(b) highlights this aspect even better by showing the different ways in which Strawman and our optimized protocols scale with an increasing number of nodes. While the overhead of Strawman increases steadily, the cost of our optimized protocols remains almost constant due to their ability to limit the propagation of unnecessary messages.

Varying the Speed of Nodes

Our second set of measurements evaluates the performance of our protocols with variable node speed. The result of an increase in node speed is clearly an increase in the frequency of reconfiguration as well as in the difficulty of reconnecting the topology when a link fails.

The results in Figure 16.2 show that the improvement achieved by our protocols over Strawman remains almost constant as node speed varies. At very low speeds, Timed Deferred Unsubscription performs better because of the WiTree protocol manages to reduce the length of the reconfiguration path; as speed increases up to $1.8m/s$, however, the optimizations exploited by Informed Link Activation allow it to achieve a better overall performance. Finally, at $2m/s$, the two protocols tend to converge to the same performance. In this case, the overlay manager may take longer to reconnect the tree, causing the subscrip-



(a) Improvement Over Strawman (b) Cost Per Reconfiguration

Figure 16.2: Improvement over Strawman and cost per reconfiguration as node speed increases.

tion timeout of INFORMED LINK ACTIVATION to elapse too soon. We believe that additional improvements can be achieved by enabling the protocol to adjust its timeouts dynamically according to the characteristics of the scenarios.

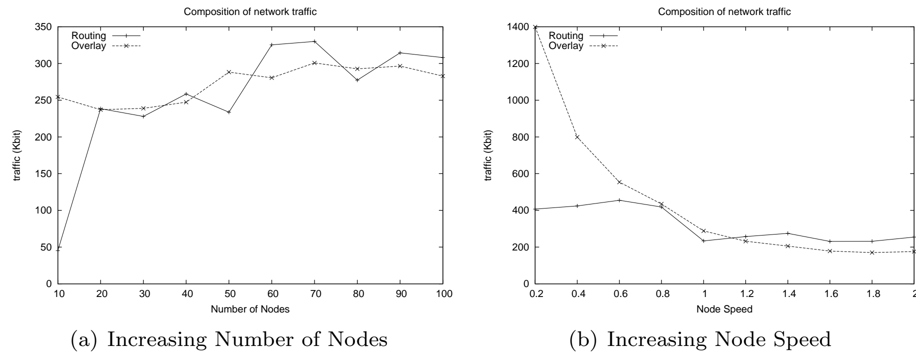
A final observation results from the analysis of the plot in Figure 16.2(b). The unit cost of reconfiguration tends to decrease as speed increases. This is because the higher frequency of reconfiguration causes several links to break simultaneously, thereby limiting the propagation of unnecessary subscriptions and unsubscriptions.

16.2.2 Routing Reconfiguration versus Overlay Maintenance

The second piece of our wireless evaluation compares the cost of reconfiguring routing information with the cost for the maintenance of the overlay. In Chapter 15, we showed that in the wired case, the cost of routing reconfiguration is by far more significant than that associated to overlay maintenance. In this section, we show that the situation is different in a wireless setting.

Similarly to what we did in the wired case, we compare the costs of the two protocols by weighting their messages according to their sizes. In the case of overlay messages we determine size according to message content, while in the case of routing messages we assume a size of 30 bytes plus headers for subscriptions, unsubscriptions and events.

The plots in Figure 16.3 show how the cost for each reconfiguration varies as the number of nodes in the network (left), or their speed (right), increases. Results show that mobility and the need to discover routes by flooding the network with request messages make the maintenance of a connected overlay in wireless networks a challenging and demanding task. With reference to the left plot, we can observe that the costs per reconfiguration of both overlay maintenance and routing reconfiguration are almost unaffected by the scale of the system. On the other hand, maintaining the overlay is more expensive when mobility is low as shown in Figure 16.3(b). The reason is that a significant portion of the traffic generated by the WiTree protocol is determined by the group hello messages broadcast by



(a) Increasing Number of Nodes

(b) Increasing Node Speed

Figure 16.3: Routing versus overlay cost for a single reconfiguration with an increasing number of nodes and with increasing node speed.

group leaders to keep group membership information up to date. When speed is high, this traffic is amortized over a large number of reconfigurations, whereas when speed is low, this traffic almost becomes a useless burden.

This suggests two ways to improve the WiTree protocol. On the one hand, the frequency of GRPH messages could be varied according to the mobility patterns of nodes. On the other hand, techniques from the LSTree protocol such as real-valued depth could be used to limit the need to propagate GRPH messages, thereby reducing the overall cost of overlay maintenance.

16.2.3 Comparison with Flooding

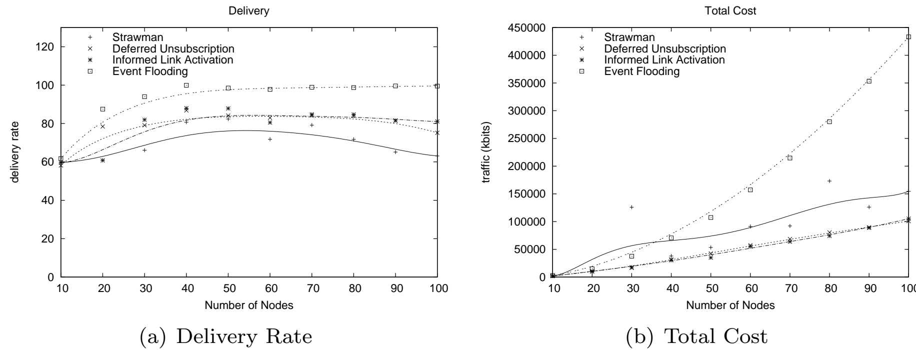
The final step in our evaluation aims to assess the validity of our approach to content-based routing in mobile ad hoc networks. As we mentioned previously, most communication solutions for MANETs exploit an unstructured approach to routing due to the inherent difficulty of maintaining correct routing information.

In this section, however, we show that a structured approach can achieve good results in wireless scenarios characterized by a low degree of mobility. Our speeds of $0.2m/s$ to $2m/s$ may correspond to the speed of people walking or to the speed of robots moving in a disaster recovery scenarios. In such cases, a tree-based routing approach results in a good delivery ratio with only a small fraction of the cost of flooding.

In the following, we present our measurements obtained in three different scenarios: with a variable number of nodes, with variable node speed and with a variable event load. In all three cases, we plot the delivery rate of our approach as well as its cost and compare it to the delivery and cost of broadcast-based event flooding.

Variable Number of Nodes

Figure 16.4 shows the results obtained with a varying number of nodes. Again, the plots show that the cost associated to our tree-based approach is significantly



(a) Delivery Rate (b) Total Cost

Figure 16.4: Delivery rate and total cost of the protocols with an increasing number of nodes.

better than that cost associated to flooding. This more and more evident as the number of nodes increases.

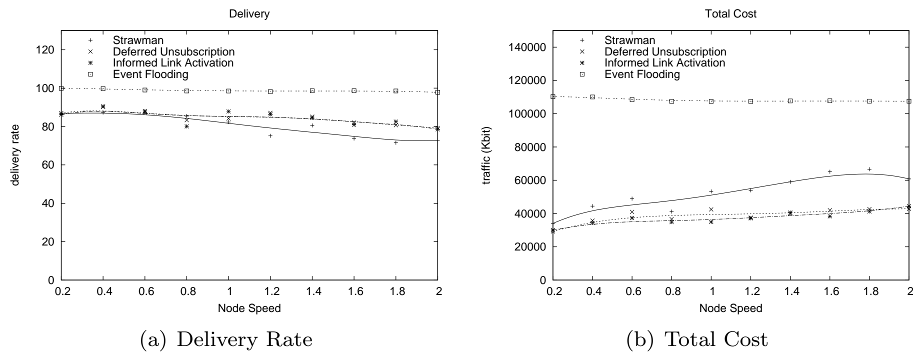
In addition, the plots show that the optimized reconfiguration protocols also achieve a significant improvement with respect to STRAWMAN both in terms of delivery rate and cost. While STRAWMAN, reaches a top delivery rate of 70%, both TIMED DEFERRED UNSUBSCRIPTION and INFORMED LINK ACTIVATION manage to deliver up 85% of events to the correct recipients.

The reason for the better delivery rate is again related to the propagation of subscriptions before unsubscriptions. The unnecessary unsubscriptions propagated by strawman cause a temporary disruption of event routes that are instead kept in place by both optimized solutions. Clearly, none of the protocols is able to reach the performance of flooding in terms of delivery; nevertheless, the applicability of the flooding approach is limited by its inherent communication cost.

Variable Node Speed

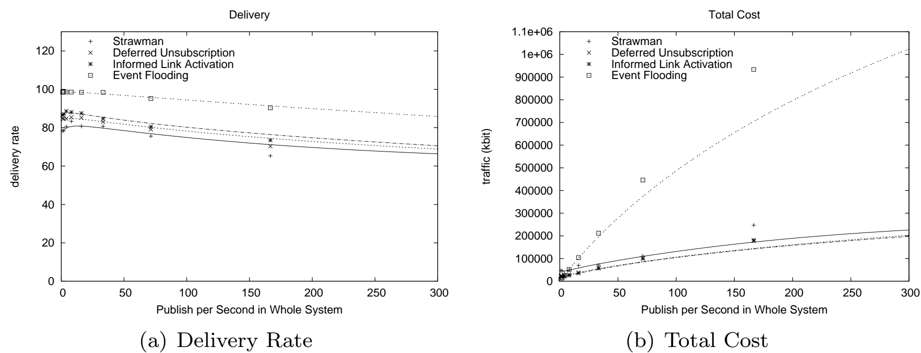
Our next set of measurements evaluates the performance of our protocols with increasing node speed. According to intuition, the ability to delivery events is negatively affected by increased node mobility. Nevertheless, our INFORMED LINK ACTIVATION and TIMED DEFERRED UNSUBSCRIPTION protocols are able to achieve an 80% delivery rate even with nodes moving at $2m/s$.

The delivery of STRAWMAN, on the other hand, drops at a faster rate and is associated to a higher communication cost. While the traffic generated by INFORMED LINK ACTIVATION and TIMED DEFERRED UNSUBSCRIPTION remains almost constant with increasing speeds, the traffic generated by STRAWMAN increases more rapidly, with the protocol generating half the messages of flooding at only $1.4m/s$



(a) Delivery Rate (b) Total Cost

Figure 16.5: Delivery rate and total cost of the protocols with increasing node speed.



(a) Delivery Rate (b) Total Cost

Figure 16.6: Delivery rate and total cost of the protocols with increasing event loads.

Variable Event Load

The final set of measurements we present is carried out with fixed speed and number of nodes and with a variable event load. According to expectations, a higher event load limits the ability of our protocols to deliver events correctly because of the higher number of collisions between packets.

The impact of collisions is visible from the fact that even the delivery of event flooding is negatively affected by high publish rates. Indeed, the difference in performance between our protocols and flooding remains almost constant. This suggests that the lower delivery associated to our protocols is more related to the inherent fragility of a tree-based routing approach than to the disruption of event routes during reconfigurations. For this reason, we are planning to improve our protocols by integrating epidemic dissemination strategies to increase their ability to route around temporary route failures resulting from the unreliability of wireless communication.

The plot in Figure 16.6(b) motivates these improvements even further by showing that the savings of our protocols in terms of total traffic increase significantly as the number of published events increases. While event flooding delivers all events to all network nodes, our protocols effectively manage to reduce traffic by contacting only a small number of nodes in addition to the intended recipients.

16.3 Concluding Remarks

The results presented in this chapter confirm the validity of the routing reconfiguration protocols developed and presented in this thesis. Both Timed Deferred Unsubscription and Informed Link Activation significantly reduce the cost of event delivery both with respect to flooding and with respect to the naive Strawman solution.

Moreover, our optimized protocols significantly improve the delivery rate of Strawman, showing that tree-based routing is an effective solution in mobile ad hoc networks when the speed of nodes remains within reasonable limits. Clearly, we aim to investigate this solution to content-based routing in MANET to a further extent both to improve its delivery rate and to reduce its cost in terms of communication.

Conclusions

The publish-subscribe communication model is enjoying increasing popularity, both in research and in 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 clients to specify these classes using linguistic facilities to match a pattern against event content.

While a number of content-based publish-subscribe systems are available, existing research efforts have mainly focused on scalability realizing the event dispatcher by means of a distributed architecture, composed of dispatching servers interconnected through an overlay network. These implementations, however, almost always fail to address issues like the dynamicity of the underlying network infrastructure.

In this thesis, we addressed this problem and presented a complete approach to the reconfiguration of content-based publish-subscribe systems. We defined an architecture for reconfigurable publish-subscribe middleware and proposed a set of new protocols for the management of the overlay and routing layers in scenarios characterized by highly dynamic topologies.

In our work, we first addressed the routing layer as routing is the defining problem in content-based publish-subscribe. Specifically, we presented a set of new reconfiguration protocols to manage the routing information enabling correct delivery of events to subscribers. When the overlay changes as a result of nodes joining or leaving the network or as a result of mobility, this information must be updated so that routing can adapt to the new environment. Our protocols manage to achieve this with as little overhead as possible.

We then addressed the overlay layer and proposed two novel approaches for building and maintaining a connected topology in highly dynamic network scenarios. Our protocols correctly achieve this goal, while at the same time managing node degree and keeping reconfigurations localized when possible. These properties make our overlay managers promising solutions not only in the context of publish-subscribe middleware but also as enabling technologies for other communication paradigms like application-level multicast.

The final part of our work consisted of an integrated evaluation of the overlay and routing layers both in wired and in wireless scenarios. Our results first show

that the optimizations provided by our routing reconfiguration protocols allow the middleware to achieve very good performance in such dynamic networks. In addition, they show that our overlay layer is able to optimize this performance even further by significantly reducing the network traffic generated by the routing layer.

The final outcome of our evaluation is therefore that efficient publish-subscribe middleware for dynamic network environments is achievable. The protocols presented in this thesis are implemented in the REDS middleware framework developed by our research group. The good performance achieved in our simulations suggests that REDS can be used to implement scalable distributed applications in large scale dynamic environments. Further research is ongoing to evaluate its performance in this type of real-world applications.

BIBLIOGRAPHY

[1] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *Proc. of the 4th Int. Wkshp. on Distributed Algorithms*, pages 15–28. Springer-Verlag New York, Inc., 1991.

[2] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, New York, NY, USA, 1999. ACM Press.

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

[4] G Antonoiu and PK Srimani. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications*, 30:1–7, 1995.

[5] Filipe Araújo, Luís Rodrigues, Jörg Kaiser, Changling Liu, and Carlos Mitidieri. Chr: A distributed hash table for wireless ad hoc networks. In *ICDCS Workshops*, pages 407–413, 2005.

[6] A. Arora and M. Gouda. Distributed reset. *IEEE Trans. Comput.*, 43(9):1026–1038, 1994.

[7] R. Baldoni, R. Beraldi, G. Cugola, M. Migliavacca, and L. Querzoni. Structure-less Content-Based Routing in Mobile Ad Hoc Networks. In *In proceedings of the International Conference on Pervasive Services (ICPS '05), Santorini, Greece, July 2005*, 7 2005.

[8] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. A self-organizing crash-resilient topology management system for content-based publish/subscribe. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, Edinburgh, Scotland, UK, May 2004.

[9] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees. In *Proc. of ACM SIGCOMM'93*, San Francisco, CA, August 1993. ACM Press.

[10] 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`.

[11] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the 19th Int. Conf. on Distributed Computing Systems*, Austin, TX, USA, 1999. IEEE Computer Society Press.

[12] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM*, pages 205–217, 2002.

[13] M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers and streaming media. In *HotNets-I, Princeton, NJ*, pages 107–112, Oct. 2002.

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

[15] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

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

[17] 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.

[18] 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.

[19] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

[20] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 163–174, New York, NY, USA, 2003. ACM Press.

[21] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.

[22] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[23] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networked Group Communication, Fifth International COST264 Workshop (NGC'2003)*, September 2003.

[24] 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.

[25] 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.

[26] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Inf. Process. Lett.*, 39(3):147–151, 1991.

[27] 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.

[28] Yan Chen, Randy H. Katz, and John Kubiatowicz. Scan: A dynamic, scalable, and efficient content distribution network. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 282–296, London, UK, 2002. Springer-Verlag.

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

[30] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12, 2000.

[31] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[32] P. Costa and D. Frey. Publish-Subscribe Tree Maintenance over a DHT. In *4th Intl. Workshop on Distributed Event-Based Systems (DEBS'05)*, Columbus, Oh, USA, June 2005. IEEE Press.

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

[34] Paolo Costa and Gian Pietro Picco. Semi-probabilistic content-based publish-subscribe. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 575–585, Washington, DC, USA, 2005. IEEE Computer Society.

[35] 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.

[36] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *19th Annual ACM Symp. on Applied Computing*, 2004.

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

[38] 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`.

[39] 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 3rd International Workshop on Software Engineering and Middleware (SEM'02), co-located with the 24th International Conference on Software Engineering (ICSE'02)*, volume 2596 of *Lecture Notes on Computer Science*, pages 187–202, Orlando (FL, USA), May 2002. Springer.

[40] Gianpaolo Cugola, Davide Frey, Amy L. Murphy, and Gian Pietro Picco. Content-based routing for publish-subscribe on a dynamic topology: Concepts, protocols, and evaluation. Technical report, Politecnico di Milano, 2005.

[41] Gianpaolo Cugola and Jose Enrique Muñoz de Cote. On introducing location awareness in publish-subscribe middleware. In *25th International Conference on Distributed Computing Systems*, Columbus, USA, 2005. IEEE Computer Society.

[42] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[43] 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.

[44] 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. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, New York, NY, USA, 2001. ACM Press.

[45] 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.

[46] 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.

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

[48] P. Francis. *Yoid Tree Management Protocol (YTMP) Specification.* ACIRI, April 2000. `http://www.icir.org/yoid/docs/ytmp.pdf`.

[49] D. Frey and A. L. Murphy. Overlay tree maintenance in large scale dynamic networks. Technical report, Politecnico di Milano, 2006. Submitted for publication. `www.elet.polimi.it/upload/frey`.

[50] J. J. Garcia-Luna-Aceves and S.Murthy. A path-finding algorithm for loop-free routing. *IEEE/ACM Trans. Netw.*, 5(1):148–160, 1997.

[51] J. Gough and G. Smith. Efficient recognition of events in distributed systems. In *Proceedings of 18th Australasian Computer Science Conference (ACSC)*, February 1995.

[52] 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.

[53] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329, New York, NY, USA, 2003. ACM Press.

[54] Cyrus P. Hall, Antonio Carzaniga, Jeff Rose, and Alexander L. Wolf. A content-based networking protocol for sensor networks. Technical Report CU-CS-979-04, Department of Computer Science, University of Colorado, August 2004.

[55] J. Hill, J. Knight, A. Crickenberger, and R. Honhart. Publish and subscribe with reply, 2002.

[56] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2):109–117, 1992.

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

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

[59] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symp. on Operating System Design and Implementation*, pages 197–212, October 2000.

[60] D.B. Johnson. Routing in Ad Hoc Networks of Mobile Hosts. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 158–163, 1994.

[61] Ion Stoica Karthik Lakshminarayanan, Ananth Rao and Scott Shenker. End-host controlled multicast routing. *Elsevier Computer Networks, Special Issue on Overlay Distribution Structures and their Applications*, 2005.

[62] 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.

[63] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, February 2003.

[64] R. Meier and V. Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications, 2003.

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

[66] Arjan J. Mooij, Nicolae Goga, and Wieger Wesselink. A distributed spanning tree algorithm for topology-aware networks. In Herwig Unger, editor, *Design, Analysis, and Simulation of Distributed systems*, pages 169–178. The Society for Modeling and Simulation International, 2004.

[67] 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`.

[68] 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.

[69] 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.

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

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

[72] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing streaming media content using cooperative networking. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 177–186, New York, NY, USA, 2002. ACM Press.

[73] 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.

[74] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *CooplS '02: Proceedings of the 7th International Conference on Cooperative Information Systems*, pages 162–173, London, UK, 2000. Springer-Verlag.

[75] R. Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. In *Proc. of the 9th Symp. on Data communications*, pages 44–53. ACM Press, 1985.

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

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

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

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

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

[81] B. Rajagopalan and M. Fairman. A new responsive distributed shortest-path rounting algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 237–246, New York, NY, USA, 1989. ACM Press.

[82] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. *Lecture Notes in Computer Science*, 2233, 2001.

[83] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[84] E.M. Royer and C.E. Perkins. Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol. In *Proc. of the 5th Int. Conf. on Mobile Computing and Networking*, pages 207–218, Seattle, WA, USA, August 1999.

[85] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, 2002.

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

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

[88] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-based content routing using xml. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 160–173, New York, NY, USA, 2001. ACM Press.

[89] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. of the ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2001.

[90] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *Int. Symp. on Software Reliability Engineering*, Paderborn, Germany, November 1998. IEEE Computer Society Press.

[91] 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.

[92] TIBCO Inc. *TIBCO Rendezvous*. `www.tibco.com`.

[93] 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.

[94] 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.

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

[96] Tak W. Yan and Hector Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, 1994.

[97] 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.

[98] 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.

[99] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1):41–53, January 2004.

[100] 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.

[101] 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.

[102] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM Press.